Persistente Datenstrukturen für asynchrone Umgebungen

am Beispiel autonomes Fahren

Bachelorarbeit

im Fachgebiet Angewandte Informatik

vorgelegt von:	Simon Danner
Studienbereich:	Angewandte Informatik
Erstgutachter:	Prof. Dr. rer. nat. Klaus Dorer
Zweitgutachter:	Prof. Dr. rer. nat. Stephan Trahasch

Kurzfassung

In der vorliegenden Bachelorarbeit wird die Verwendung von persistenten Datenstrukturen in Anwendungen untersucht. Dafür werden zunächst verschiedene Eigenschaften von bestehenden Implementierungen persistenter Datenstrukturen im Vergleich zu ephemeralen Implementierungen der Datenstrukturen untersucht. Hierbei werden Laufzeiten und Speicherverbrauch analysiert. Dabei wird gezeigt, in welchen Fällen die Verwendung von persistenten Datenstrukturen Vorteile gegenüber der Verwendung ephemeralen Datenstrukturen bringt.

Im zweiten Teil wird die Verwendung von persistenten Datenstrukturen in asynchronen Umgebungen untersucht. Hierfür werden die grundlegenden Probleme aufgezeigt, die bei der Erzeugung von Thread lokalen Versionen geteilter Zustände entstehen. Es wird aufgezeigt, wie die verschiedenen Versionen durch ein Zusammenführungsverfahren in einen gemeinsamen Zustand überführt werden können. Als konkrete Implementierung wurde eine Framework in Java entwickelt, welches versucht dieses Problem wiederverwendbar zu lösen. Das entwickelte Framework wird anhand einer Beispielsdomäne evaluiert, indem die Performanz mit der Verwendung von expliziter Synchronisation verglichen wird. Hierbei wird gezeigt, dass in Situationen in denen die Modifikationen viel Zeit benötigen, der Ansatz des Zusammenführen Vorteile gegenüber expliziter Synchronisation bietet.

Im letzten Teil der Arbeit wird die Verwendung eines geteilten persistenten Zustandes im an der Hochschule Offenburg durchgeführtem A_2O Projekt untersucht, bei dem Software für eine autonomes Modellauto entwickelt wurde. Es werden die Schwierigkeiten bei der Umsetzung der im Framework entwickelten Techniken in C++, sowie mögliche Lösungswege aufgezeigt.

Vorwort

Ich möchte an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelor-Arbeit unterstützt haben.

Ganz besonders gilt dieses Dank Prof. Dr. rer. nat. Klaus Dorer, der meine Arbeit und somit auch mich betreut hat. Vor allem möchte ich für die Betreuung des A_2O Teams während der Vorbereitungen auf den Audi Autonomous Driving Cup danken. Durch die tatkräftige Unterstützung konnte ich während des Projektes einiges lernen, und durch die gute Organisation hat die Teilnahme am Wettbewerb viel Spass gemacht.

Daneben gilt mein Dank Prof. Dr. rer. nat. Stephan Trahasch, der bereit war die Arbeit als Zweitbetreuer zu betreuen.

Nicht zuletzt gebührt meiner Familie Dank, die mich während des gesamten Studium unterstützt hat. Gerade in den letzten Monaten war diese Unterstützung für mich sehr wichtig.

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelor-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Offenburg, den 7. Juli 2015

Simon Danner

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Als persistente Datenstrukturen werden Datenstrukturen bezeichnet bei denen alte Versionen weiter bestehen. Das heißt alle Versionen der Datenstrukturen können weiter verwendet werden.

1.1 Ziel der Arbeit

Im Rahmen der Arbeit soll die Verwendung von persistenten Datenstrukturen in asynchronen Umgebungen evaluiert werden. Auftretende Probleme sollen aufgezeigt werden, sowie Lösungen oder Lösungsansätze hierfür entwickelt werden. Das Thema soll sowohl in einer Java Umgebung als auch in C++ untersucht werden.

1.2 Aufbau der Arbeit

In Grundlagen Persistente Datenstrukturen werden zunächst Grundbegriffe geklärt und die verschiedenen Arten persistenter Datenstrukturen vorgestellt. Danach werden in Persistente Datenstrukturen Frameworks in Java / C++ verschiedene Projekte vorgestellt, die Implementierungen von persistenten Datenstrukturen in Java und C++ bereitstellen.
In Kapitel Anwendung und Performanz Persistenter Datenstrukturen wird die Performanz einiger dieser Implementierungen evaluiert und eine Beispielsdomäne gezeigt und untersucht, in der eine persistente Datenstruktur verwendet wird.
Ab dem Kapitel Persistente Datenstrukturen in asynchronen Umgebungen geht es um die Verwendung von persistenten Datenstrukturen in asynchronen Umgebungen. Hier werden zunächst die zu lösenden Probleme asynchroner Umgebungen beschrieben und wie die Verwendung von persistenten Datenstrukturen zur Lösung der Probleme beitragen kann.
In Entwickeltes Framework zur Verwaltung von geteilten Zuständen wird das im Rahmen der Bachelorarbeit entwickelte Framework vorgestellt. Mit ihm können Zuständen zusammengeführt werden, die in asynchronen Umgebungen von einem gemeinsamen Vorgängerzustand abgeleitet wurden. Es werden die Probleme die bei der Umsetzung aufgetreten sind dargestellt, die Implementierung beschrieben und

eine Evaluation der Ergebnisse durchgeführt.
Im Folgenden Kapitel, Audi Autonomous Driving Cup, wird zunächst der Rahmen des Audi Autonomous Driving Cup vorgestellt. Hierbei handelt es sich um einen Wettbewerb, bei dem Studententeams Software für autonome Modellautos entwickelten. Danach wird auf die Architektur der vom A2O Team entwickelten Software eingegangen und beschrieben wie persistente Datenstrukturen in dem Projekt verwendet werden können. Am Schluss folgt noch ein Ausblick auf weiter zu untersuchende Probleme und offene Fragestellungen, die nicht in der Bachelorarbeit behandelt werden.

2 Grundlagen Persistente Datenstrukturen

Um die nachfolgenden Definitionen und Erklärungen einfacher gestalten zu können, werden zunächst einige Grundbegriffe definiert und erklärt.

2.1 Datenstruktur

Als Datenstruktur bezeichnet man eine Ansammlung von Daten die Operationen bereitstellt, die auf ihr ausgeführt werden können. Die Operationen lassen sich dabei in Query Operationen und Update Operationen einteilen. Query Operationen greifen auf die Daten nur lesend zu und ändern sie nicht, während Update Operationen die in der Datenstruktur enthaltenen Daten verändern.

2.2 Immutable Datenstrukturen

Als immutable Datenstrukturen oder deutsch unveränderbare Datenstrukturen werden Datenstrukturen bezeichnet, auf denen keine Update-Operationen ausgeführt werden können. Sie lassen sich nach dem Anlegen nicht modifizieren. Zunächst erscheint dies als eine unpraktische Einschränkung. Diese Eigenschaft macht es jedoch möglich in bestimmten Situationen einfachere und korrektere Programme zu schreiben.

In den folgenden Situationen bieten unveränderbare Datenstrukturen gegenüber veränderbaren oft Vorteile:

- Schreibzugriff verbieten:
  Es ist oft sinnvoll bestimmte Daten als Schutz vor Modifikationen als Nur-Lesen-Sicht an andere Komponenten eines Programms zu geben. Wenn diese nur Lesezugriff benötigen, kann damit sichergestellt werden, dass diese Komponenten keine Möglichkeit haben die Daten zu ändern. Durch das anlegen einer Nur-Lesen-Sicht gibt es die Möglichkeit, das Verhalten im Programm festzulegen und nicht nur als Konvention oder in einem Style Guide.

- Parallele Verwendung:
  Da sich immutable Datenstrukturen nicht ändern, kann bei der parallelen Verwendung in mehreren Threads auch ohne explizite Synchronisation davon ausgegangen werden, dass die einzelnen Datenfelder insgesamt einen validen Zustand repräsentieren. Außerdem ist garantiert, dass ein unveränderbares Objekt über die ganze Lebenszeit die gleichen Werte beinhaltet. Durch die Vermeidung von expliziter Synchronisation und Sicherheitskopien kann oft die Performance verbessert werden.

- Verwendung als Feld:
  Wenn Objekte in einem Feld eines anderen Objektes gespeichert werden sollen, muss oft darauf geachtet werden, dass das gespeicherte Objekt danach nicht mehr geändert wird. Deswegen wird oft eine Defensiv-Kopie ausgeführt. Bei nicht-änderbaren Objekten fällt dies weg.

- Zusammengesetzte Zustände:
  Wenn Objekte als Repräsentation von Zuständen dienen, ist es oft der Fall, dass mehrere Teilwerte zusammen den Zustand darstellen. Dabei ist der Zustand durch die Werte zu einem bestimmten Zeitpunkt definiert. Bei Änderungen muss deshalb darauf geachtet werden, dass diese atomar durchgeführt werden. Sonst könnte ein anderer Thread mit einem invaliden Zustand arbeiten.

2.2.1 Immutable Datenstrukturen in Java

Unveränderbare Objekte können in Java realisiert werden. Einige Java Standardklassen sind bereits als unveränderbare Klassen implementiert. Die wohl bekanntesten sind java.lang.String und die Wrapper-Klassen der primitiven Typen. Beispielsweise java.lang.Short und java.lang.Boolean.
Es gibt allerdings keine Unterstützung der Sprache die es ermöglicht festzulegen, dass Instanzen einer Klasse unveränderbar sind. Um unveränderbare Objekte zu implementieren empfiehlt es sich daher einige Konventionen einzuhalten. Der Entwickler muss selbst darauf achten, dass eine immutable Klasse auch beim Hinzufügen oder Ändern von Feldern und Methoden immutable bleibt. Für die Umsetzung sollten folgende Regeln befolgt werden, die auch in der offiziellen Java Dokumentation aufgelistet sind [Siehe: Oracle, Java Tutorial]:

1. Felder final und privat:
   Felder sollten nicht von außen überschrieben werden können.

2. Vererbbarkeit verbieten:
   Entweder sollte die Klasse mit final markiert werden, oder die Konstruktoren

als private definiert werden. Über statische Factorymethoden können die Objekte erzeugt werden. Dadurch kann von der Klasse nicht geerbt werden. Das verhindert das Überschreiben von Methoden in Unterklassen. Diese könnten ermöglichen Felder zu überschreiben.

3. Objekte in einem Aufruf konstruieren:
   Dadurch gibt es zu keinem Zeitpunkt invalide Objekte, die nur teilweise konstruiert sind.

4. Keine Update Methoden:
   Es darf keine Methoden geben die Felder des Objektes verändern.

5. Defensiv Kopien bei veränderbaren Feldern:
   Wenn das Klasse veränderbare Objekte als Feld enthält, muss sowohl bei Rückgabe des Feldes (z.B. getter-Methode) als auch beim Konstruieren des Objektes eine Kopie angelegt werden. Sonst könnte das immutable Objekt mit einer Referenz zu einem mutablen Objekt konstruiert werden. Wenn dann zu einem späteren Zeitpunkt das referenzierte Objekt geändert wird, würde auch das als immutable gedachte Objekt geändert.

2.2.2 Immutable Datenstrukturen in C++

Ähnlich wie bei Java gibt es auch in C++ einige Dinge, die beachtet werden müssen, um ein unveränderbares Objekt zu implementieren.

1. Wie auch bei Java gelten die Regeln Immutable Datenstrukturen in Java Punkt 2 bis Immutable Datenstrukturen in Java Punkt 5.

2. Auch Punkt 1 muss in angepasster Form beachtet werden. In C++ werden unveränderbare Member mit dem Schlüsselwort const markiert. Seit C++11 können auch in C++ Klassen mit dem final Schlüsselwort markiert werden, was erbende Klassen verhindert.

3. Das C++ Schlüsselwort const kann dazu benutzt werden unveränderbare Objekte anzulegen. Auf Objekten welche einen const Qualifier besitzen können nur Methoden aufgerufen werden, welche auch mit const markiert wurden. In diesen Methoden können keine Member modifiziert werden. Dadurch werden bei Änderungen am Code durch Entwickler, die versuchen das Objekt in diesen Methoden zu modifizieren, Verletzungen der Unveränderbarkeit schon durch den Compiler erkannt.
   Allerdings können auf mit const markierten Objekten nur const markierte Methoden aufgerufen werden. Der Vorteil des const Schlüsselwort ist, dass Instanzen beliebiger Datentypen unveränderbar gemacht werden können. Die

explizite Möglichkeit der Markierung hilft dem Programmierer beim Nachvollziehen der Funktionsweise des Codes, da er davon ausgehen kann, dass const Objekte nicht modifiziert werden.

2.2.3 Ephemerale Datenstrukturen

Eine ephemerale Datenstruktur ist eine Datenstruktur, von der es nach einer Update-Operation nur eine gültige Version gibt. Eine Datenstruktur welche die Daten im Zustand vor der Update Operation enthält, existiert nicht mehr. Datenstrukturen die nicht persistent sind, sind automatisch ephemerale Datenstrukturen.

2.2.4 Persistente Datenstrukturen

Das Wort Persistenz wird in der Informatik oft im Zusammenhang mit dem Abspeichern von Daten auf Speichermedien benutzt. Hierbei ist persistentes speichern als Gegensatz zum halten von Daten in flüchtigem Speicher, wie im RAM, zu sehen.
Bei Datenstrukturen bedeutet persistent allerdings nicht, dass diese persistiert werden können im Sinne von 'auf die Festplatte schreiben'. Persistente Datenstrukturen sind Datenstrukturen, die bei Modifikationen ihre vorherige Version bestehen lassen. Dies bedeutet, dass nach einer Update Operation sowohl die geänderte Version, als auch die bisherige Version weiterhin benutzt werden können. Bei der Implementierung von persistenten Datenstrukturen muss daher bei Update-Operationen darauf geachtet werden, dass die zu ändernde Version nicht modifiziert wird.

Mit Hilfe verschiedener Techniken kann die Persistenz Eigenschaft umgesetzt werden. Um dies zu bewerkstelligen bieten sich verschiedene Implementierungswege an. Die einfachste Variante ist das tiefe Kopieren der Ursprungsversion und die anschließende Modifikation der Kopie. Das tiefe Kopieren bedeutet, dass sowohl das Objekt selbst, als auch die Objekte die von ihm referenziert werden rekursiv kopiert werden müssen. Da oft nur kleine Teile von Objekten verändert werden, ist dieses Vorgehen allerdings sehr unperformant bezüglich des Speicherverbrauchs und der Laufzeit. Das einfache Kopieren kann verbessert werden, indem unveränderte Bereiche wiederverwendet werden.

Persistente Datenstrukturen können anhand von Garantien bezüglich der Veränderbarkeit vorheriger Versionen und anderer Eigenschaften kategorisiert werden. Im Folgenden soll kurz die Bedeutung der verschiedenen Begriffe erläutert werden.

2.2.4.1 Partiell persistente Datenstrukturen

Bei partiell persistenten Datenstrukturen können auf vorherigen, sowie der aktuellen Version Query Operationen ausgeführt werden. Update Operationen können jedoch nur die aktuelle Version verändern. Durch diese Einschränkung entsteht eine linear geordnete Versionshistorie, bei der neue Versionen der Datenstruktur aus der aktuellen Version erzeugt werden können.

2.2.4.2 Voll persistente Datenstrukturen

Bei voll persistenten Datenstrukturen können sowohl Query, als auch Update-Operationen auf allen alten sowie der aktuellen Version ausgeführt werden. Dadurch entsteht ein Versionsbaum, bei dem die Versionen der Datenstrukturen die Knoten darstellen.

2.2.4.3 Confluently persistente Datenstrukturen

Confluently persistente Datenstrukturen wurden von Driscoll et. al in Driscoll u. a. [1991] als vollpersistente Datenstrukturen die Update Methoden bereitstellen, bei denen neue Versionen aus zwei alten Versionen erstellt werden können beschrieben. Diese Operationen werden Merge oder Meld Operationen genannt.

Da jede Version von mindestens einer, teilweise aber auch mehreren Vorgängerversionen abgeleitet wurde, entsteht ein gerichteter azyklischer Graph.

2.2.4.4 Rein-funktionale persistente Datenstrukturen

Als funktionale Datenstrukturen werden Datenstrukturen bezeichnet, deren Werte sich nicht verändern lassen. Solche Datenstrukturen werden oft als Standarddatenstrukturen in funktionalen Programmiersprachen wie Clojure oder Haskell verwendet. Rein-funktionale persistente Datenstrukturen sind somit gleichzeitig immutable und persistent.

Bei funktionalen persistenten Datenstrukturen können somit keine Modifikationen an bestehenden Objekten ausgeführt werden. Da keine Objekte geändert werden können ist auch garantiert, dass bestehende Versionen nicht zerstört werden. Daher sind funktionale persistente Datenstrukturen automatisch voll persistent.

Auch wenn sie selbst immutable sind, stellen sie trotzdem Änderungsoperationen zur Verfügung. Diese geben ausgehend von dem bisherigen Objekt ein anderes Objekt mit der gewünschten Änderung zurück.

2.2.4.5 Transiente Datenstrukturen

In vielen Anwendungen werden auf Datenstrukturen viele Update-Operationen in einer einzelnen Methode ausgeführt. Beispiele sind Initialisierungen oder Modifikations-Operationen, die einen großen Anteil der von der persistenten Datenstruktur enthaltenen Objekte modifizieren.
In solchen Fällen wäre es gut, wenn diese Modifikationen mit einer Version der Datenstruktur ausgeführt werden würden, in der die Operationen performanter ausgeführt werden können. Ein Ansatz diese Problem zu lösen wäre das Kopieren einer persistenten Datenstruktur in eine ephemerale Datenstruktur. Dies hätte eine Komplexität von $O(n)$, abhängig von der Größe der Datenstruktur.
Transiente Datenstrukturen versuchen dieses Problem zu lösen, in dem sie die Elemente der persistenten Datenstruktur verwenden. Sie werden dazu aus einer Version einer persistenten Datenstruktur erzeugt. Dann können die gewünschten Update-Operationen ausgeführt werden und die transiente wieder in eine persistente Form zurückgeführt werden. Durch das Verwenden von Strukturen der ursprünglichen persistenten Version kann das Kopieren vermieden werden. Transiente Datenstrukturen sind also ephemerale Datenstrukturen, die aus persistenten Datenstrukturen erzeugt werden und wieder in persistente Versionen zurückgeführt werden können.

2.2.5 Implementierung persistenter Datenstrukturen

Für die verschiedenen Arten von persistenten Datenstrukturen wurden verschiedene Verfahren entwickelt, mit denen sie realisierbar sind.

2.2.5.1 Partielle persistente Datenstrukturen

Jede Datenstruktur, die eine konstante Anzahl von Verweisen auf jeden Knoten besitzt, kann in eine partiell persistente Datenstruktur umgewandelt werden.

Dies wurde in "Making data structures persistent" von Driscoll u. a. [1986] nachgewiesen. Dabei ist der Overhead, der durch eine Modifikation der modifizierte Datenstruktur erzeugt wird, für Zeit und Speicherplatz ein konstanter Faktor. Das bedeutet, dass viele Datenstrukturen wie verkettete Listen oder B-Bäume mit den beschriebenen Methoden transformiert werden können und dabei für viele Anwendungsfälle performant genug sind.
Die einfachste mögliche Implementierung einer partiell persistenten Datenstruktur stellt der Ansatz der "Fat Nodes" dar. Hierbei werden sowohl die aktuelle Version als auch die vorherigen Versionen der Daten in dem gleichem Knoten gespeichert.

Eine Fat Node beinhaltet Felder, in denen die Ursprungswerte beim Anlegen des Knotens enthalten sind. Für jedes Feld gibt es eine Liste mit den auf dem Feld angewandten Modifikationen. Die Modifikationen bestehen aus einem Paar von neuem Feldwert (das Resultat einer Update-Operation), sowie einem Zeitstempel. Bei einer Modifikation eines Feldes wird der neue Wert zusammen mit einem Zeitstempel in die Modifikationsliste eingefügt. Wenn auf ein Feld in einer bestimmten Version mittels Zeitstempel zugegriffen werden soll, kann durch ein Vergleich der Zeitstempel in der Modifikationsliste der zum gewünschtem Zeitpunkt enthaltene Wert des Felds gefunden werden. Da hier eine Binärsuche verwendet werden kann, besteht für das Finden des gewünschten Wertes eine Komplexität von $O(\log m)$ bei Leseoperationen, wobei m die Anzahl der Modifikationen ist. Die Komplexität der Update- und Leseoperationen lässt sich verbessern, indem nicht alle Modifikationen im gleichen Knoten gespeichert werden, sondern nur eine konstante Anzahl.

In Driscoll u. a. [1986] wird dieses verbesserte Verfahren als ”Node copying” beschrieben. Hierbei wird bei einer Update-Operation auf einem Knoten ein neuer Knoten angelegt, wenn die Modifikationsliste voll ist. Der neue Knoten beinhaltet die geänderte Version des Feldes sowie eine leere Modifikationsliste. Dieser neue Knoten muss in allen Knoten, die auf die ursprüngliche Version zeigten, in die Modifikationsliste mit aufgenommen werden. Falls auch bei diesen Knoten die Modifikationsliste voll ist, müssen auch für diese neue Knoten angelegt werden. Dieses Verfahren muss bis zum Ursprungsknoten durchgeführt werden. Unter der Annahme, dass die Anzahl der eingehenden Verweise konstant ist, wird eine amortisierte Komplexität von $O(1)$ erreicht.

2.2.5.2 Path Copying

Path Copying ist ein anderes Verfahren, mit dem Persistenz erreicht werden kann. Hierbei wird bei jeder Update-Operation eine modifizierte Kopie des geänderten Knotens angelegt. Danach muss für jeden Knoten, der auf die Ursprungsversion zeigte, eine Kopie angelegt werden, welche auf den nachfolgenden modifizierten Knoten zeigt. Dadurch entsteht ein neuer Pfad, während der alte unmodifiziert bestehen bleibt. Es muss zusätzlich ein Array von Wurzelknoten angelegt werden, indem Verweise auf die Versionen sowie ein Timestamp abgelegt werden. Über dieses Array kann über den Timestamp auf beliebige Versionen zugegriffen werden.

Die Komplexität für den Zugriff auf eine beliebige Version wird durch die benötigte Binärsuche auf dem Array mit Verweisen auf die Wurzelknoten $O(\log m)$.

2.2.5.3 Kombination Fat Node / Path Copying

Sleator, Tarjan et al. stellen in Driscoll u. a. [1986] eine Verfahren vor, welches durch eine Kombination des Fat Node und des Path Copying Verfahrens Verbesserungen ermöglicht. Im sogenannten Node-Splitting Verfahren werden im Falle, dass ein neuer Knoten angelegt werden muss, weil die Modifikationsliste voll ist, in den neuen Knoten genau wie beim 'Fat Node' Verfahren der neue Werte geschrieben. Zusätzlich wird die Hälfte der Modifikationsliste des vollen Knoten in den neuen Knoten verschoben. Dies führt dazu, dass sowohl in der Ursprünglichen Version als auch in der neuen Version wieder Platz für zusätzliche Versionen entsteht. Allerdings steigt die Komplexität der Implementierung, da von jedem Knoten aus Verweise auf Vorgängerversionen benötigt werden, und in manchen Fällen Verweise auf Knoten in alten Versionen verändert werden müssen.

2.2.5.4 Voll persistente Datenstrukturen

Bei Voll-persistenten Datenstrukturen entsteht anstatt einer linearen Versionshistorie ein Versionsbaum. In Driscoll u. a. [1986] wird eine Variante des Fat Node Verfahrens vorgestellt, mit dem volle Persistenz erreicht wird. Hierfür wird bei Update Operationen nicht immer an die Modifikationsliste angehängt, sondern die Verweise auf neue Knoten in einer Liste hinter die Elternversion eingefügt, auf der die Modifikation ausgeführt wurde. Query und Modify-Operationen müssen leicht abgeändert werden, um die korrekte Version in den für volle Persistenz angepassten Modifikationslisten zu finden.

2.2.5.5 Confluently persistente Datenstrukturen

Um Confluently Persistente Datenstrukturen umzusetzen, kann die Methode des Path Copying, die in Reps u. a. [1983] beschrieben ist verwendet werden. Hierbei muss festgelegt werden, welche Pfade von welcher Version übernommen werden.

2.2.6 Anwendungsbeispiele in der Praxis

In den folgenden Abschnitten sollen kurz Projekte vorgestellt werden, die persistente Datenstrukturen benutzen.

2.2.6.1 Basisdatenstruktur für Editor

Vis ist ein experimenteller konsolenbasierter, vim ähnlicher Texteditor, welcher eine persistente Datenstruktur als Kern verwendet. Die verwendete Datenstruktur ist eine Implementierung der persistenten Piece Table Datenstruktur, welche in Crowley [1998] beschrieben wird. Diese wird auch in anderen Editoren und Textverarbeitungsprogrammen wie AbiWord verwendet. Hierbei werden Änderungen am bearbeiteten Text durch das Einfügen neuer Knoten abgebildet. Da die ursprünglichen Versionen noch gültig sind, sind für Texteditoren typische Operationen wie das Rückgängig machen durchgeführter Änderungen einfach zu implementieren.

Der Quellcode ist unter einer Open Source Lizenz unter Tanner verfügbar.

2.2.6.2 Verwendung in Funktionalen Programmiersprachen

Als funktionale Programmiersprachen werden Programmiersprachen bezeichnet, welche das Programmierparadigma des funktionalen Programmieren unterstützen. Hierzu zählen unter anderem Lisp, Clojure, Scala, F# und Haskell.

In diesen Programmiersprachen implementierte Programme strukturieren ihren Code in Funktionen. Hierbei werden die vom Programm durchzuführenden Berechnungen als Auswertungen von Funktionen modelliert. Ein Funktion ist hierbei als mathematische Funktion zu sehen, die keine Nebeneffekte erzeugt. Sie wird also für die gleichen Parameter immer das selbe Ergebnis zurückliefern. Durch diesen Ansatz können Effekte wie Zustandsübergänge und veränderbare Datenstrukturen vermieden werden. Dadurch wird es für den Programmierer einfacher den Code zu analysieren, da Nebeneffekte ausgeschlossen werden können.

Um dieses Verhalten garantieren zu können, werden in einigen Implementierungen von Programmiersprachen und ihren Standardbibliotheken voll persistente, unveränderbare Datenstrukturen benutzt. Dadurch kann sichergestellt werden, dass eine Funktion keine Zustandsänderung an diesen Datenstrukturen verursachen kann. Da wichtige Datenstrukturen wie Vectoren oder Maps oft verwendet werden, versuchen die Programmiersprachen möglichst performante Implementierungen einzusetzen.

Scala und Clojure verwenden beispielsweise für ihre Implementierungen eines Vectors persistente Datenstrukturen, welche auf dem Paper von Bagwell Bagwell [2001] aufbauen.

3 Persistente Datenstrukturen Frameworks in Java / C++

Um die Vorteile persistenter Datenstrukturen nutzen zu können, gibt es verschiedene Bibliotheken, welche persistente Implementierungen grundlegender Datenstrukturen wie Listen, Maps oder Sets bereitstellen. Im Folgenden sollen einige Implementierungen persistenter Datenstrukturen für die Sprachen Java und C++ vorgestellt werden.

3.1 Java

In Java gibt es einige wenige Implementierungen persistenter Datenstrukturen. Die beiden Projekte PCollections und clj-ds stellen beide nicht nur Implementierungen einzelner Datenstrukturen bereit, sondern versuchen die am häufigsten von Programmierern benutzten Datenstrukturen in einer persistenten Java-Umsetzung bereitzustellen. Im Folgenden werden die beiden Projekte und ihre Implementierungen kurz vorgestellt.

3.1.1 clj-ds

Clojure ist eine funktionale Programmiersprache, welche in Java implementiert ist und auf der Java Virtual Machine läuft. Ein Bestandteil von Clojure sind Datenstrukturen, welche funktional und persistent sind. Um die Datenstrukturen auch innerhalb von reinen Java Programmen nutzen zu können, wurden sie von Karl Krukow aus der Clojure Codebase extrahiert. Zwar ist auch die in der Clojure Codebase vorhandene Implementierung in Java geschrieben, die Klassen besitzen jedoch Abhängigkeiten zu anderen Teilen von Clojure. Die Extrahierung ermöglicht es die Datenstrukturen als Bibliothek in eigenen Projekte zu verwenden. Dabei müssen keine zusätzlichen Clojure Bibliotheken und kein Clojure Compiler benutzt werden.

Der Source Code des Projektes befindet sich auf Github unter Krukow.

clj-ds beinhaltet folgende Datenstrukturen:

PersistentArrayMap
: Persistente Implementierung einer Map, die intern Arrays verwendet und dadurch lineare Zugriffszeiten ermöglicht, allerdings wird bei Änderungsoperationen das Array komplett kopiert, wodurch das Hinzufügen von Elementen teuer ist.

PersistentHashMap
: Persistente Implementierung eines Hash Array Mapped Trie nach Phil Bagwell Bagwell [2001]. Benutzt das Verfahren Path Copying.

PersistentHashSet
: Implementierung eines Sets, welches intern eine Persistente HashMap benutzt.

PersistentHATTrie
: Implementierung eines HAT Trie nach Askitis und Sinha [2007], welche Path Copying verwendet. Diese Datenstruktur ist nicht in Clojure vorhanden und wurde vom Autor des clj-ds Projektes implementiert. Hat Tries können für Mappings von Strings zu Werten benutzt werden.

PersistentList
: Implementierung einer verketteten Liste, ohne Remove und ähnlichen Operationen.

PersistentQueue
: Persistente Implementierung eines Queues. Basierend auf Batch Queues Okasaki [1999].

PersistentTreeMap
: Persistente Implementierung eines Red Black Trees.

PersistentTreeSet
: Persistente Implementierung eines TreeSets, nutzt intern eine PersistentTreeMap.

PersistentVector
: Persistente Implementierung eines Vectors. Basierend auf Ideen aus Bagwell [2001]. Benutzt das Path Copying Verfahren, was auf einem Baum mit großer Breite angewandt wird. Durch die große Breite des Baumes kommt es nur zu geringen Tiefen wodurch die Performanz verbessert wird.

Diese Klassen sind alle voll persistent implementiert. Wie in der Aufzählung ersichtlich wird, wurden viele Ideen und Konzepte aus Papers der letzten Jahre aufgegriffen und in eine benutzbare Implementierung gebracht. Teilweise können Operationen auf Objekten dieser Klassen transiente Objekte zurückgeben, welche es ermöglichen performanter Änderungen durchzuführen. Diese sind jedoch nicht persistent, sollten

also nur in abgeschlossenen Bereichen verwendet werden. Die transienten Klassen sind:

- TransientArrayMap
- TransientHash

Update-Operationen auf ihnen besitzen eine Komplexität von $O(1)$. Die Klassen implementieren die Collection Interfaces von Java, so dass sie einfach in bestehenden Code verwendet werden könnten. Allerdings werden viele Methoden nicht unterstützt, beim Aufruf dieser wird eine UnsupportedOperationException geworfen.

3.1.2 PCollections

PCollections ist wie clj-ds eine Sammlung von Klassen, welche die üblichen Collection Interfaces in Java in einer persistenten Art implementieren. Die Bibliothek stellt folgende Klassen bereit:

- HashTreePMap
  Stellt eine Implementierung einer HashMap bereit. Baut auf einen Größenbalancierten Binärbaum nach Adams [1993] auf.
- HashTreePSet
  Stellt eine Implementierung eines Sets bereit. Benutzt intern eine HashTreePMap.
- HashTreePBag
  Stellt eine Implementierung eines Bags bereit. Bags sind Sets, bei denen mehrere gleiche Objekte beinhaltet werden können. Benutzt intern eine HashTreePMap.
- ConsPStack
  Stellt eine Implementierung eines Stacks bereit. Die Elemente werden hierbei einer verketten Liste angeordnet, wodurch Persistenz einfach zu erreichen ist.
- TreePVector
  Stellt eine Implementierung eines Vectors bereit. Intern wird wie bei der HashTree Klasse ein größenbalancierter Binärbaum benutzt.

Die implementierten Klassen sind voll persistent implementiert. Ein besonderer Fokus von PCollections liegt auf der Interoperabilität zu den Standard Java Collections. So implementieren alle Klassen von PCollections das Interface java.util.Collection. Die einzelnen Klassen implementieren außerdem die entsprechenden Interfaces aus java.util. So kann zum Beispiel jede Instanz einer PCollections HashTreePMap wie

eine Implementierung des java.util.Map Interfaces behandelt werden. Dies ermöglicht es bestehenden Code einfach für die Verwendung von PCollections anzupassen.

3.2 C++

Auch in C++ gibt es einige wenige Open Source Projekte die versuchen Datenstrukturen persistent zu implementieren.

Ein grundsätzliches Problem bei der Implementierung in C++ besteht in der Speicherverwaltung. Gegenüber Sprachen mit Garbage Collection wie Java, `C#` oder Python muss der Speicher in C++ manuell verwaltet werden. Da bei persistenten Datenstrukturen versucht wird möglichst viele Objekte für neu erzeugte Versionen wiederzuverwenden, ist es wichtig, dass Objekte welche noch referenziert werden nicht gelöscht werden.

3.2.1 Speicherverwaltung C++

Um Unterobjekte über die Lebenszeit eines Objektes bestehen zu lassen, müssen diese in C++ dynamisch allokiert werden. Wenn Unterobjekte statisch angelegt werden, werden sie automatisch beim deallokieren des beinhaltenden Objektes freigegeben. Dynamisch allokierte Objekte müssen allerdings auch um Speicherlecks zu vermeiden wieder freigeben werden, wenn sie nicht mehr benötigt werden.
In C++ wird eine dynamische Allokation durch den new Operator ausgeführt. Als Resultat wird ein Pointer auf das durch den Konstruktor initialisierte Objekt zurückgegeben. Um das Objekt zu deallokieren, muss der Operator delete aufgerufen werden, welcher den Dekonstruktor des Objektes ausführt und den allokierten Speicher des Objektes wieder freigibt.
Bei der Verwendung von Pointern in C / C++ ist es oft nicht klar welche Komponente für die Verwaltung des Speichers und das Löschen des Objektes zuständig ist. Dieses Wissen ist allerdings nötig, um Objekte zum korrekten Zeitpunkt freizugeben. Traditionell wurden diese Informationen durch Kommentare oder Konventionen festgelegt. Wenn das Freigeben des Speichers nicht korrekt implementiert wird, kommt es zu sogenannten mehrfachen Deallokations (double free) Fehlern welche zu Progammabstürzen und Sicherheitsproblemen führen können.
Dies führt auch zu Problemen, wenn dynamisch allokierte Objekte von mehreren Objekten referenziert werden und erst gelöscht werden sollen, wenn sie nicht mehr benötigt werden.
Anwendungen welche die manuelle Speicherverwaltung vereinfachen wollten, mussten lange eigene Verfahren entwickeln, mit denen festgestellt werden kann, wann

es nötig und erlaubt ist ein Objekt zu löschen. Die Grundlage der meisten Umsetzungen ist die Annahme, dass Objekte die nicht mehr referenziert werden gelöscht werden können. Viele Bibliotheken verwenden daher intern ein Reference Counting genanntes Verfahren. Beim Referenz Counting wird in jedem Objekt ein Zähler mitgeführt, welcher die Referenzen auf das Objekt mitzählt. Durch das Überschreiben von Operatoren wie dem Zuweisungs- oder dem delete Operator kann in C++ genau ermittelt werden, wie oft ein Objekt referenziert wird. Der delete Operator wird nur dann aufgerufen, wenn dieser Zähler auf 0 dekrementiert wird. Diese Technik wird auch von der im AADC Projekt genutzten Bildverarbeitungsbibliothek OpenCV in den Kern Matrix Klassen, der PointCloud Library in Form von Boost Smart-Pointern sowie auch in AADC eigenem Code benutzt.

Mit dem C++11 Standard wurden neue Mechanismen geschaffen die explizit festlegen welche Komponente für das Verwalten des Speichers zuständig ist. Die in den Standard aufgenommenen Smart Pointer kapseln rohe Pointer und nehmen die Objekte auf die sie zeigen in Besitz. Auf die gekapselten Objekte kann durch die üblichen C++ Pointer Operatoren zugegriffen werden. Allerdings kann auf Smart Pointern keine Pointer Arithmetik ausgeführt werden, da sie im Hintergrund als normale C++ Objekte implementiert sind, welche nicht die gleichen Garantien wie herkömmliche C/C++ Pointer erfüllen.

Die folgenden Pointertypen sind ab dem C++11 Standard im std Namespace Header 'memory' enthalten:

- unique_ptr
  Ist nicht kopierbar und besitzt alleiniges Eigentum über das Zielobjekt. Wenn der unique_ptr aus dem Scope fällt, wird das verwaltete Objekt zerstört. Dies ermöglicht es zum Beispiel explizit bei der Rückgabe eines Pointers festzulegen, dass der Caller nun der Eigentümer des Objektes ist, was bei C++ vor C++11 oder C nicht möglich ist.

- shared_ptr
  Zeigt auf ein Objekt, auf dass optional auch von anderen shared_ptr gezeigt wird. Wenn der letzte shared_ptr zerstört wird, wird auch das Objekt zerstört.

- weak_ptr
  Verweist nicht besitzend auf ein Objekt, welches von einem shared_ptr verwaltet wird. Dies kann zum Aufbrechen von zyklischen shared_ptr Referenzen verwendet werden. Sie können auch dazu verwendet werden Objekte zu benutzten, die von shared_ptr verwaltet werden, ohne den Zeitpunkt der Zerstörung der Objekte zu beeinflussen. Bei einem Zugriff wird der Pointer temporär zu einem shared_ptr umgewandelt. Dieser stellt sicher, dass das verwaltete Objekt, solange es über diesen Pointer verwendet wird, nicht gelöscht wird.

Der Standard legt hierbei wie bei C++ üblich nur die Typen, das Verhalten und die Methoden der Klassen fest. Wie die Vorgaben implementiert werden, hängt von der verwendeten Standard Bibliothek ab. Für die Implementierung von shared_ptr wird jedoch meist das Referenz Counting Verfahren verwendet.

3.2.2 Okasaki Projekt - Bartosz Milewski

Bartosz Milewski hat mehrere persistente Datenstrukturen in modernem C++ umgesetzt. Die Konzepte stammten hierfür größtenteils aus Okasaki [1999]. Umgesetzt wurden von ihm Implementierungen von Listen, Queues, Maps und Red-Black Trees. Für die Umsetzung werden die oben genannten Smart Pointer genutzt. Der unter einer Public Domain Lizenz stehende Code ist als Demonstration einer Implementierung dieser Datenstrukturen in modernem C++ zu sehen, nicht als aktives oder fertiggestelltes Projekt. Der Code ist unter Milewski verfügbar.
Da einige Features von C++14 wie Reverse Iterators verwendet werden, muss der Code mit einem Compiler, sowie einer Standardbibliothek die C++14 unterstützt kompiliert und gelinkt werden oder für ältere Compiler und Bibliotheken angepasst werden. Diese Unterstützung ist momentan nur in der Standardbibliothek des llvm Projektes, libc++, in einer stabilen Version implementiert. Die Verwendung solcher aktuellen Standards macht die Verwendung in Projekten die mehrere Plattformen und Compiler unterstützen schwierig.

Die einzelnen Implementierungen werden hierbei jeweils in eigenen Headern die unabhängig voneinander verwendet werden können bereitgestellt. Die Klassen stellen außerdem einige Funktionen bereit, durch die funktionale Programmierung mit ihnen möglich wird. So gibt es Funktionen wie fold (auch reduce genannt), welche Funktionen entgegennehmen und diese auf die Inhalte der Datenstruktur anwenden.

3.2.3 Gorgone - Ignacio Blasco López

Ignacio Blasco López portierte die Implementierung des persistenten Vectors, wie er in Clojure verwendet wird, zu C++. In der Portierung wird die smart Pointer Implementierung der Boost Bibliothek benutzt, welche als Vorlage für die in C++11 enthaltenen Smart Pointers gilt. Falls der PersistentVector in einem Projekt ohne boost Abhängigkeit benutzt werden soll, könnte einfach auf die C++11 Variante umgestellt werden. Auch hier ist die Portierung nicht als aktives Projekt anzusehen. Die Implementierung ist in einer einzelnen Datei verfügbar und steht unter der Eclipse Public Lizenz. Der persistente Vektor ist generisch implementiert, so dass

er für beliebige Typen verwendet werden kann. Der Code ist unter PersistentVector verfügbar.

4 Anwendung und Performanz Persistenter Datenstrukturen

In diesem Kapitel soll die Verwendung von persistenten Datenstrukturen untersucht werden. Dafür werden Performanz-Messungen durchgeführt und mit anderen Implementierungen von Datenstrukturen verglichen.

4.1 Java Implementierungen

Um die verschiedenen Implementierungen der Java Bibliotheken zu bewerten, wurden einfache Tests ausgeführt. Bei ihnen wird die Zeit gemessen, die benötigt wird vorgegebene Anzahl von Operationen auf diesen Datenstrukturen auszuführen. Alle Messungen wurden mit einem PC durchgeführt, der mit einer Intel® Core™ i5-2520M CPU und 8 GB RAM ausgestattet ist. Als JVM wurde OpenJDK in Version 8.u45 mit Standard Einstellungen unter Linux benutzt.

Zunächst wurde die ArrayList aus der Java Standard-Bibliothek mit den entsprechenden Datenstrukturen von Closure-ds sowie PCollections verglichen.

In Abbildung 4.1 wurde die Laufzeit gemessen die benötigt wird, um eine vorgegebene Anzahl von Elementen einem Vektor Objekt der jeweiligen Implementierung hinzuzufügen. Es fällt auf, dass die benötigte Zeit beim TreePVector der PCollection Bibliothek linear ansteigt. Bei der PersistentVector Klasse bleibt die Kurve dagegen näher an der effizientesten Implementierung, der ArrayList der Java Standard-Bibliothek. In der Praxis würde die Anwendung eines TreePVector bei Programmen, die eine große Menge von Elementen in einem Vektor halten müssen zu Performanceeinbußen führen. Der PersistenteVector von clj-ds zeigt auch bei großen Mengen einzufügender Elemente geringe Unterschiede gegenüber der Implementierung der Standardbibliothek. Obwohl er persistent implementiert ist, ist er ähnlich performant wie die ephemerale ArrayListe. Dies zeigt, dass auch persistente Datenstrukturen effizient implementiert werden können. Allerdings wird auch aufgezeigt, dass die Performanz sehr von der konkreten Implementierung abhängt.

In Abbildung 4.2 wurden die Laufzeiten verglichen die benötigt wird, um die in den Vectoren gespeicherten Elemente abzurufen. Dabei wurde aus einem bereits gefüllten

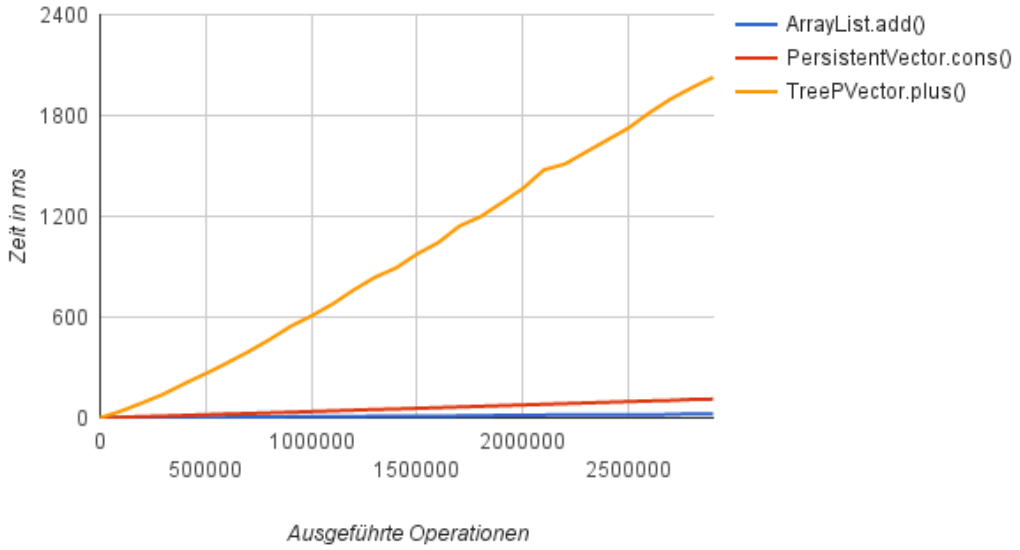


Abbildung 4.1: Vergleich Laufzeiten add Operation Vector-Klassen

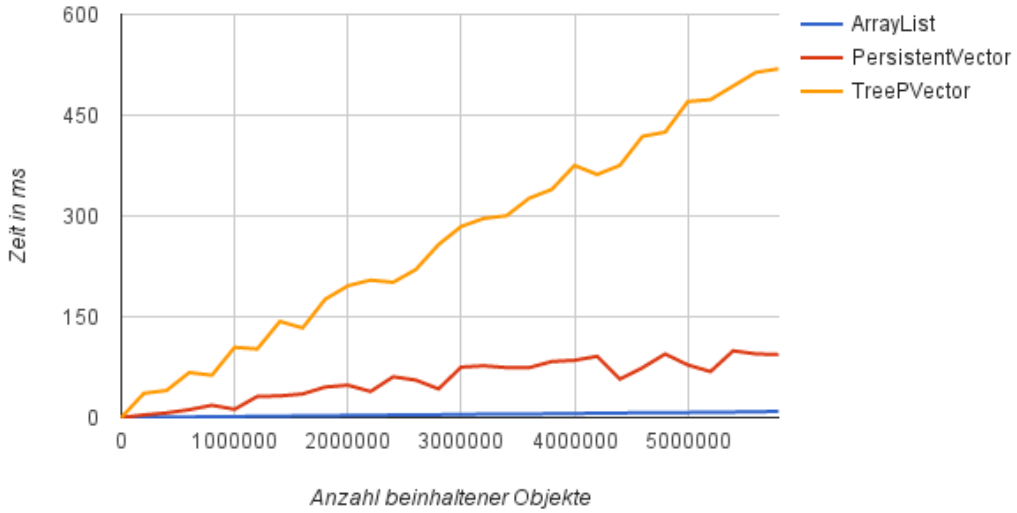


Abbildung 4.2: Vergleich Laufzeiten get-Operation Vector-Klassen

Vektor jedes Element einmal abgerufen. Bei der get-Operation ist die ArrayListe am performantesten, während der TreePVector linear mehr Zeit braucht.

Als zweite Betrachtung wurden die gleiche Messungen mit Implementierungen von HashMaps durchgeführt. Auch beim dem Vergleich der HashMap Klassen, java.util.HashMap, PersistentHashMap von clj-ds und HashTreePMap werden große Unterschiede er-

sichtlich. In 4.3 werden die Laufzeiten der add Operationen der Map Klassen verglichen. Hierbei wurden verschiedene Anzahl von Objekten eingefügt und die Zeit gemessen.

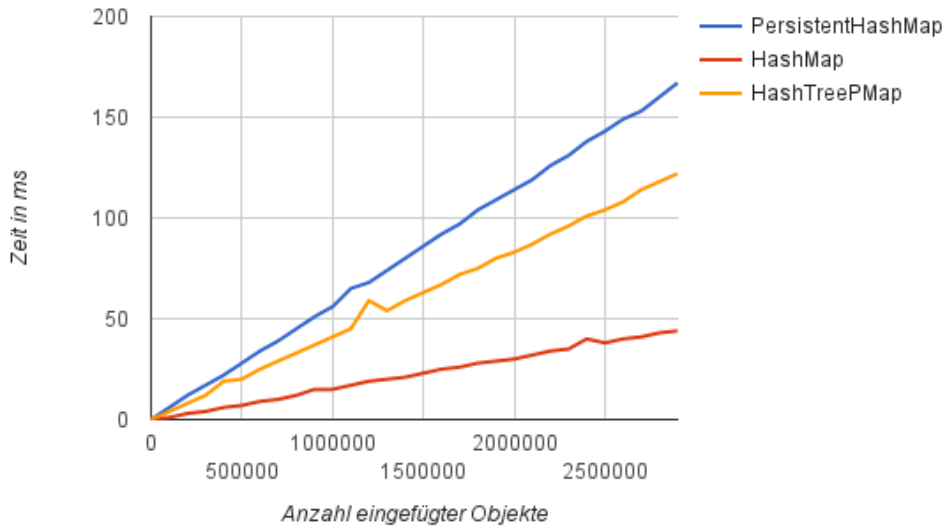


Abbildung 4.3: Vergleich Laufzeiten add-Operation Map-Klassen

Wieder ist die Implementierung der Standardbibliothek die schnellste. Hier gewinnt jedoch die Implementierung von PCollections, die HashTreePMap gegenüber der clj-ds Implementierung PersistentHashMap.

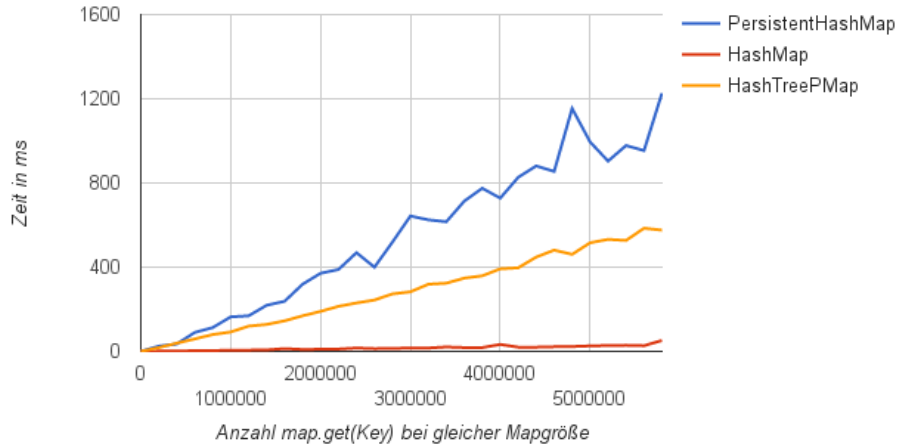


Abbildung 4.4: Vergleich Laufzeiten get(Key)-Operation Map-Klassen

Mit bereits gefüllten HashMaps wurde ein Vergleich der Laufzeiten beim Abrufen der Werte gemessen. Auch in 4.4 ist die Implementierung von PCollections performanter gegenüber der clj-ds Implementierung. Allerdings steigt bei beiden die benötigte Laufzeit linear mit der Größe der HashMap, während die Laufzeit bei der ephemeralen HashMap der Standardbibliothek annähernd gleich bleibt.

Es ist zu beobachten, dass die persistenten Implementierungen gegenüber den ephemeralen Implementierungen der Standardbibliothek langsamer arbeiten. Dies war jedoch zu erwarten, da die Erhaltung der Persistenz Eigenschaft notwendigerweise zusätzliche Komplexität mit sich bringt. Interessant ist, dass die beiden Projekte jeweils Implementierungen bereitstellen, die schneller sind als die entsprechende Implementierung des anderen Projektes. Hier wäre es schön, wenn ein Projekt die Ideen des anderen aufgreifen könnte, damit ein Anwendungsentwickler nicht selbst bei jeder Datenstruktur Messungen durchführen muss, um die performanteste persistente Implementierung verwenden zu können. Außerdem muss bei der Benutzung der Datenstrukturen darauf geachtet werden, ob ihre Performanz im Gesamtprogramm relevant ist. Gerade bei großen Sammlungen mit vielen Elementen sind die Laufzeitunterschiede gegenüber anderen Implementierungen sehr groß. Für viele Anwendungen werden allerdings die Vorteile der persistenten Datenstrukturen überwiegen, da oft nicht die Performanz der Datenstrukturen der limitierende Faktor ist.

4.2 Persistent Vector C++

Für den Performance Vergleich der C++ Implementierung des persistenten Vectors aus dem Gorgone Projekt, werden wie bei dem Vergleich der Java Implementierungen Vektoren mit verschiedener Anzahl von Elementen befüllt und die dafür benötigte Zeit gemessen.

Die Messungen wurden auf dem gleichen Rechner ausgeführt. Als Compiler wurde clang++ in Version 3.6.1 verwendet. Kompiliert wurde mit dem Optimierungslevel -O 2. Benutzt wurde die Version 1.58 von boost.

In Abbildung 4.5 werden die Laufzeiten bei insert Operationen am Ende des Vektors im PersistentVector sowie in std::vector verglichen. Im Vergleich zu den Daten bei den Java Implementierungen fällt hier auf, dass die persistente C++ Implementierung deutlich langsamer als die Implementierung der Standardbibliothek ist. Das ist überraschend, da die C++ Implementierung eine recht direkte Portierung des Java Codes von clj-ds darstellt.

Mit dem Systemprofiler Zoom von rotateright wurde ein Durchlauf des Programmes analysiert. Hierbei wurde durch Sampling mit einer Rate von einem Sample pro Mil-

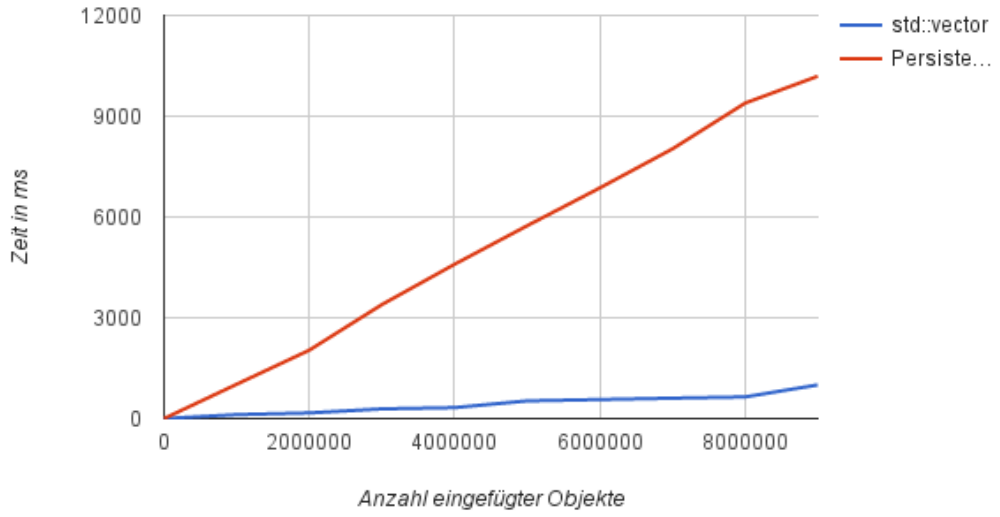


Abbildung 4.5: Vergleich Laufzeiten PersistentVector, std::vector

Methodenname	Anteil an Gesamtlaufzeit
boost::detail::sp_counted_impl_p::Branch>::~sp_counted_impl_p()	18.5%
PersistentVector::add(block const&)	12.8%
PersistentVector::Leaf::~Leaf()	11.7%
boost::detail::sp_counted_impl_p::Leaf>::~sp_counted_impl_p()	7.8%
PersistentVector::Branch:: Branch()	6.7%
_int_free	2.9%
PersistentVector::popRoot(...)	1.9%
PersistentVector::findTail()	1.3%
PersistentVector::pushTail(...)	0.8%

Tabelle 4.1: Anteil von Methoden an der Gesamtlaufzeit bei PersistentVector.insert()

lisekunde ermittelt, in welcher Methode die meiste Zeit verbracht wird. In Tabelle 4.1 werden die Methoden mit dem größten prozentualen Anteil an der Gesamtlaufzeit aufgelistet. Die in der Tabelle mit Farbe unterlegten Methoden sind Destruktor Methoden. Das bedeutet, dass ein Großteil der Laufzeit mit dem Zerstören nicht mehr benötigter Objekte verbracht wird.
Dies ist bei std::vector nicht der Fall, da dieser wenn hinzugefügt wird keine alten Objekte freigibt. Der Persistente Vektor muss jedoch durch das Path Copying Verfahren alle Knoten freigeben, die auf dem Pfad vom neu hinzugefügten Wert bis zum Root Knoten liegen. Da im Test immer der aktuelle persistente Vector mit einem modifizierten überschrieben wird, werden die veränderten Knoten der alten Version

bei jeder Änderung freigegeben. Falls die Ursprungsversion weiterhin referenziert werden würde, wäre dies nicht der Fall und die Destruktoren würden nicht aufgerufen werden. Beim Einfügen vieler Werte ist der Persistente Vector also durch die zusätzlich benötigte Speicherverwaltung deutlich langsamer.

4.3 Cleaner

Die Cleaner Domäne ist eine Beispielsdomäne, anhand derer verschiedene Suchalgorithmen bezüglich der Laufzeit sowie des Speicherverbrauches evaluiert werden können.

Die Domäne wird in 'Artificial Intelligence: A Modern Apporach' von Russell und Norvig [2010] beschrieben, und als einfache Beispielsdomäne verwendet. Die verwendete Implementierung in Java wurde von Prof. Dr. Klaus Dorer umgesetzt.

Die Cleaner Domäne bildet eine Kette von Räumen ab, welche einen Zustand besitzen, der festlegt ob der Raum momentan sauber oder verschmutzt ist. Der Zustand aller Räume sowie die Position eines Saugers werden zusammen in einem VacuumWorldState gekapselt. Aus einem VacuumWorldState kann mithilfe von Operatoren ein neuer State erzeugt werden. Diese Operatoren stellen mögliche Schritte zum erreichen des Zieles alle Räume sauber zu bekommen dar. Das Ziel ist es einen Zustand zu erhalten, bei dem alle Räume gesäubert sind. Die Suchalgorithmen müssen dafür Zustände auswählen, von denen aus sie durch die Anwendung der Operatoren die weiteren möglichen Zustände untersuchen. Wenn ein Zustand erreicht wird der die Zielbedingung erfüllt, kann die Suche beendet werden.

Die einzelnen Zustände werden als Knoten eines Baum repräsentiert, welche es ermöglichen Suchalgorithmen wie Tiefensuchen und Breitensuche anzuwenden, die Informationen zur Position des Zustandes im Möglichkeitsbaum benötigen. Es gibt drei Operatoren mit denen ein neuer geänderter Zustand aus einem Zustand abgeleitet werden kann. Diese sind Left, Right und Clean. Bei Left wird der virtuelle Sauger einen Raum nach links bewegt, bei Right einen Raum nach rechts. Bei Clean wird der Raum gesäubert, in dem der Sauger momentan steht.

Durch Änderungen an der internen Repräsentation der Zustände kann der Einfluss von Änderungen an den zu Grunde liegenden Datenstrukturen gemessen werden.

4.3.1 Umsetzung Mutable / Tiefes kopieren

Bei der Umsetzung mit Kopieren werden bei der Anwendung der Operatoren neue Zustände erzeugt, indem der gesamte Ursprungs-Zustand kopiert wird. Der Status

der Räume wird als Bitset gespeichert, das für jeden Raum ein Bit enthält. Ist ein Bit nicht gesetzt, ist der Raum sauber. Bei Anwendung eines Operators wird das gesamte Bitset kopiert. Im Falle des Suck-Operators wird außerdem das betreffende Bit genullt. Kopiert wird durch aufrufen der clone() Methode des Bitsets, welche das gesamte Bitset-Objekt dupliziert. Die Position des Saugers wird als Integer-Wert gespeichert, welche zum Index des Raumes im Bitset entspricht.

4.3.2 Umsetzung Partly-Persistent

Die ursprüngliche Version des VacuumWorldState nutzt bereits Unveränderbarkeit an einer Stelle, um den Speicherverbrauch zu minimieren und unnötiges kopieren zu vermeiden. Dies geschieht wenn ein neuer Zustand erzeugt wird, in dem nur die Position des Saugers verändert wird. Wenn also ein Left- oder Rightoperator benutzt wird, kann das Bitset des Vorgängerzustand referenziert werden, da dieses nur im Konstruktor modifiziert wird und somit sicher unveränderbar ist.

4.3.2.1 Umsetzung Persistent

Um die VacuumWorldState Klasse voll persistent umzusetzen, wird anstelle der java.lang.BitSet Klasse eine Umsetzung verwendet, welche intern den Persistenten Vectore des clj-ds Projektes benutzt. In dieser Klasse werden mehrere BitSets in einem PersistentenVector abgelegt. Die Größe der einzelnen BitSets kann konfiguriert werden. Die Klasse implementiert die von der Domäne benötigten Methoden des java.lang.BitSets, sodass es für diesen Anwendungsfall wie ein Standard Bitset benutzt werden kann. Dabei werden Bitset-Indizes auf die im Vector enthaltenen Bitsets umberechnet.

Durch die Benutzung des Persistenten Vectors können außer den Bewegungsoperationen auch die Sauge-Aktion realisiert werden, ohne das gesamte Bitset zu kopieren. Anstatt bei einer Set-Aktion ein neues Bitset zu erstellen, muss nur eines der Bitsets im Persistenten Vector durch ein neues ersetzt werden. Die anderen Zustände besitzen immer noch ihre Version des Bitsets, während der persistente Vektor in der neuen Version ein geändertes Bitset enthält.

Dieses Vorgehen verringert den Speicherverbrauch, durch Vermeidung von unnötigem kopieren des Bitsets.

4.3.3 Performance Vergleich

Um die verschiedenen Umsetzungen vergleichen zu können, wurde das Programm bezüglich Laufzeit und Speicherverbrauch mit unterschiedlichen Problemgrößen gemessen.

Die Suche wird dabei durch einen A*-Stern Algorithmus durchgeführt. Alle Messungen wurden mit einem PC ausgeführt, der mit einer Intel® Core™ i5-2520M CPU und 8 GB RAM ausgestattet ist. Um vergleichbare Ergebnisse zu erhalten, wurde die Prozessorgeschwindigkeit auf 2.50GHz festgelegt. Die Software lief dabei unter Linux auf der OpenJDK Java Virtual Machine, Version 7u71. Die Java VM wurde mit den Argumenten -Xms4512M -Xmx4512M gestartet. Diese Optionen geben die initiale beziehungsweise die maximale Größe des von der VM verwendeten Speicher Allokationpools an. Dies ist nötig, da aufgrund des hohen Speicherverbrauchs der Anwendung mit den Standardwerten nur Messungen für sehr kleine Problemgrößen möglich sind.

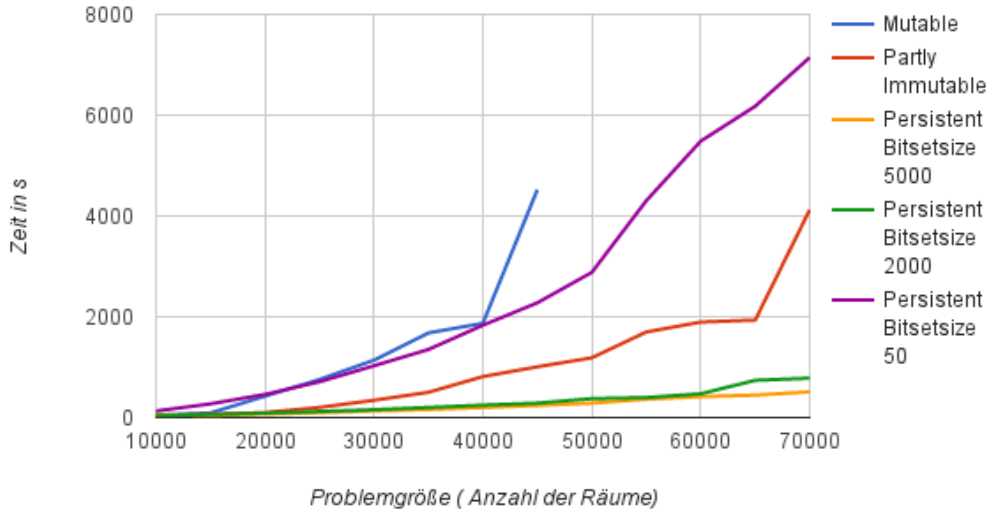


Abbildung 4.6: Vergleich Laufzeiten

Beim Betrachten der Laufzeiten bezüglich der Problemgrößen in 4.6 fallen zunächst die steilen Anstiege auf. Diese existieren bei der Kopier- sowie bei der Partly Immutable Variante. Dies ist mit der Speicherverwaltung von Java zu erklären. Da die Kopieren Variante für jeden erzeugten State ein neues Bitset benötigt, steigt ihr Speicherverbrauch am schnellsten. Ab einer Problemgröße von ca. 40.000 Räumen steigt die Laufzeit stark an. Dies liegt an dem Garbagecollector der Java Virtual Machine, der immer öfter versucht unbenutzten Speicher zu ermitteln und diesen freizugeben.

Da jedoch die einzelnen States noch benötigt werden und den Großteil des benutzten Speichers ausmachen, kann der Garbage Collector kaum Speicher freigeben. Das führt zu vielen Aufrufen des Garbage Collectors, bevor bei einer Problemgröße von ca. 43.000 nicht mehr genügend Speicher allokiert werden kann. Dadurch stürzt das Programm mit einer Out of Memory Exception ab.

Das gleiche Verhalten kann auch bei der Partly Immutable Implementierung beobachtet werden. Allerdings wird hier durch das Wiederverwenden der Bitsets, beim Erzeugen neuer Zustände aus den Right und Left Operatoren, Speicher gespart. Deswegen können mit dieser Variante größere Probleme bearbeitet werden. Hier tritt das beschriebene Verhalten ab einer Problemgröße von ca. 65.000 Räumen auf. Bei der Verwendung von persistenten Datenstrukturen tritt dieses Verhalten nicht auf, da hier weniger Speicher benötigt wird.

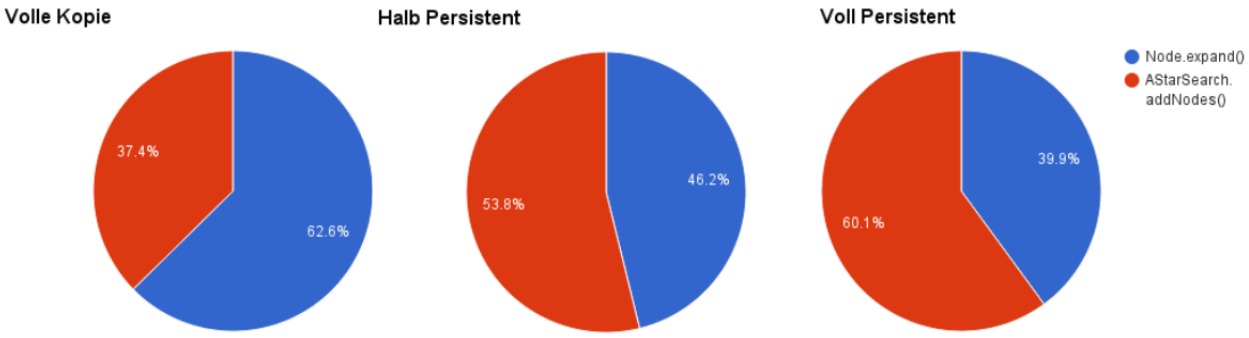


Abbildung 4.7: Vergleich Zeit Verbrauch

In Abbildung 4.7 wird dargestellt, welchen Methoden die meiste Laufzeit in den verschiedenen Ansätze benötigen. Dabei wurde für ein Problem von 40.000 Räumen die Anteile der Methoden an der Laufzeit pro Verfahren gemessen. Die meiste Laufzeit kann zwei Methoden zugerechnet werden, welche klar getrennte Aufgaben haben. Da Zustandsänderungen nur in einer Methode ausgeführt werden, können die Unterschiede zwischen den einzelnen Implementierungen klar aufgezeigt werden. In der Node.expand() Methode werden neue Zustände erstellt, indem alle drei Operatoren auf einen Ursprungszustand angewendet werden. Die Zustände werden, wie oben beschrieben, mit unterschiedlichen Verfahren erzeugt. In der AStarSearch.addNodes() werden die neu erzeugten Zustände zur Liste der von der A*-Suche zu verarbeitenden Zustände hinzugefügt. Dabei wird eine Beurteilung der Zustände durch die A*-Suche durchgeführt. Dafür wird eine Schätzfunktion verwendet, welche in diesem Beispiel verhältnismäßig recht aufwendig ist. Der Code der Methode ist für alle Implementierungen gleich.

Die Grafik zeigt den prozentualen Anteil der beiden Methoden an der Gesamtlaufzeit. Man kann erkennen, dass der Anteil an der Gesamtlaufzeit der Methode zum

Neuerzeugen von Zuständen bei der Methode in der das gesamte BitSet kopiert wird, mit 62.6% am höchsten ist. In der Variante in der nur bei dem Saug-Operator das BitSet kopiert wird, ist der Anteil schon bedeutend geringer, da nur 1/3 der BitSets gegenüber dem ersten Ansatz kopiert werden. In der voll persistenten Implementierung beträgt der Anteil nur noch 39.9%. Hier werden keine Kopien der gesamten BitSets angelegt, nur noch einzelne kleine BitSets kopiert, modifiziert und in den persistenten Vektor eingearbeitet.

Durch die Verwendung von Persistenten Datenstrukturen kann in diesem Fall also die Performanz durch das Vermeiden von Kopien verbessert werden.

Da für eine Problemgröße von ca. 43.000 bzw. 65.000 Räumen bei den ersten beiden Implementierungen kein freier Speicher mehr vorhanden ist, kann das Problem mit diesen Implementierungen und JVM Einstellungen nicht gelöst werden. Hier kommt ein Vorteil der persistenten Implementierung zum Tragen. Sie ermöglicht die Lösung von größeren Problemen, was mit den anderen Ansätzen auf Grund des Speicherverbrauches unmöglich war.

Eine weiteren Parameter der untersucht werden kann, ist die Größe der BitSets, welche in dem persistenten Vector gespeichert werden. Bei größeren BitSets wird bei Änderungen an einem Bit mehr Speicher benötigt, da ein neues ganzes BitSet für die Änderung angelegt wird. Allerdings steigt bei kleineren BitSets die Zeit die benötigt wird um Operationen auf dem BitSet auszuführen. In der Beispielsdomäne ist dies vor allem die Methode BitSet.previousSetBit. Bei dieser wird das zu dem Index gehörende BitSets berechnet, und in diesem das vorherige gesetzte Bit gesucht. Falls dieses nicht in diesem BitSet gefunden werden kann, muss das vorherige BitSet untersucht werden. Diese relative einfache Operation nimmt bei großen Problemstellungen mit kleinen unterliegenden BitSets ein signifikanten Anteil an der Laufzeit ein.

Ein Vergleich der Laufzeiten ist in 4.8 dargestellt. Hierbei wurde mit verschiedenen Größen des BitSets Messungen durchgeführt. Die Messungen für jeden Versuch enden bei den Problemgrößen bei denen es zu einem Programmabruch wegen fehlendem Speicher kam. Hier bestätigt sich, dass die Verwendung von großen BitSets zu einem höheren Speicherverbrauch führt. So können mit einer BitSet Größe von 5000 nur Probleme mit bis zu 600.000 Räumen bearbeitet werden. Mit BitSets die weniger Elemente enthalten, können größere Probleme bearbeitet werden. Hier sinkt aber die Performanz, da Zugriffe auf die BitSet Strukturen langsamer werden. Der Speicherverbrauch sinkt bei BitSet-Größen unter 1000 nicht mehr signifikant, ist bei 5000 Räumen aber noch deutlich höher, was sich in einem frühen Programmabruch äußert. In der Messung brechen bei einer Problemgröße von 90.000 die Implementierugen mit einer Bitset Größe von 100, 500, 2000 ab.

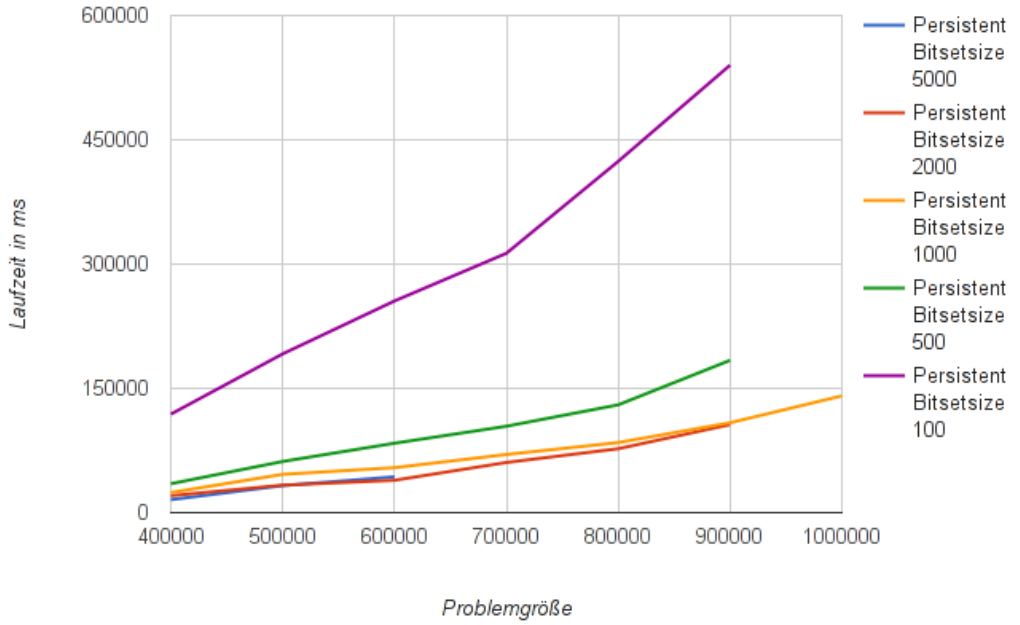


Abbildung 4.8: Laufzeitvergleich Cleaner Domäne Persistente Implementierung

Dieses Verhalten ist durch den ähnlichen Speicherverbrauch zu erklären. Bei diese Problemgröße wird das Programm mit einer Exception des Garbage Collectors abgebrochen. Der Garbage Collector Overhead wird zu groß und das Programm beendet. Mit einer Bitset Größe von 1000 lässt sich allerdings noch ein Problem der Größe 1.000.000 lösen. Dies liegt wohl an der Implementierung des Garbagecollectors, der bei dieser Größe des Bitsets und damit der spezifischen Anzahl erzeugter Objekte besser umgehen kann und nicht in den Grenzfall des zu hohen Overheads läuft.

4.3.3.1 Fazit

Die Verwendung des persistenten Vectors erlaubt es in dieser Domäne Probleme zu lösen, die mit den anderen Ansätzen nicht lösbar waren. Vor allem der geringere Speicherverbrauch gegenüber von Kopien ermöglicht dies. Da die Zugriffsoperationen des Vectors effizient implementiert sind, fällt die aufwändigere Zugriffsoperation gegenüber des Standard Bitsets in der Gesamtlaufzeit nicht negativ ins Gewicht.

Außerdem ist ersichtlich, dass die Wahl der Größe der zu Grunde liegenden Strukturen, welche bei Änderungsoperationen verändert und somit kopiert werden, großen Einfluss auf das Verhalten des Programmes nimmt.

5 Persistente Datenstrukturen in asynchronen Umgebungen

In asynchronen Umgebungen, in denen Code von mehreren Ausführungssträngen gleichzeitig ausgeführt wird, entstehen in objektorientierten Sprachen Probleme. Wenn mehrere Threads zur gleichen Zeit mit den gleichen Objekten arbeiten, kann es zu sogennanten Data Races kommen. So können Membervariablen die von mehreren Threads nebenläufig modifiziert werden, vom Programmierer nicht erwartete Werte enthalten. Diese Probleme können durch unterschiedliche Vorgehensweisen vermieden werden.

Die wohl am weitesten verbreitete Methode sichert den betroffenen Code durch explizite Synchronisierung ab. Diese stellt sicher, dass immer nur ein Ausführungsstrang zu einem Zeitpunkt ein Objekt manipuliert. Ein anderer Ansatz ist das vollständige vermeiden von geteilten Zustandsvariablen. Wenn doch Zustandsvariablen geteilt werden sollen, können Probleme durch das Teilen von unveränderbaren Objekten vermieden werden.

5.1 Probleme in asynchronen Umgebungen

Die Verwendung von expliziter Synchronisation führt bei falscher Anwendung zur Entstehung von anderen Fehlern. So kann es zu Dead-Locks kommen, bei denen beispielsweise Thread A auf Thread B wartet, welcher jedoch auf Thread A wartet, wodurch die Ausführung des Programmes stoppt.

In Farchi u. a. [2003] wird ein Versuch unternommen, auftretende Fehler in Java bei der Verwendung von Multithreading in Kategorien zu klassifizieren. Sie kategorisieren hierbei Fehler in die folgenden drei Oberklassen:

- Code von dem angenommen wird er sei geschützt:
  Bei vielen Fehlern wird fälschlich davon ausgegangen, dass der betroffene Code nicht von threading Problemen betroffen ist. Dadurch wird die Verwendung der expliziten Synchronisation vergessen.
  Wenn synchronisiert wird, können die Bereiche nicht weit genug gefasst sein.

Ein weiterer Problemfall ist es, wenn Annahmen über die Atomarität von Operationen getroffen werden die nicht zutreffen.

- Inkorrekte Annahme des Programmablaufes
  Falsche Annahmen zum Programmablauf können zu Problemen führen. So können Benachrichtigungen zwischen Threads verloren gehen, wenn niemand auf sie wartet. (wait() / notify() in Java)
- Geblockte oder tote Threads
  Unerwartetes blockieren von Threads oder Threads die nicht mehr kontrolliert werden können, führen zu Fehlern im System.

5.2 Verwendung von Persistenten Datenstrukturen

Durch die Unveränderbarkeit von funktionalen persistenten Datenstrukturen bieten diese sich an, um Zustandsobjekte zwischen Threads zu teilen.

Durch die Unveränderbarkeit können mehrere Threads Update-Operationen auf den geteilten Objekten ausführen, ohne dabei Data Races zu verursachen. Außerdem muss keine explizite Synchronisation verwendet werden, wodurch ganze Fehlerklassen wie fehlende Synchronisation oder inkorrekte Synchronisation vermieden werden. Allerdings ist zu beachten, dass jeder Thread nach einer Update-Operation einen eigenen lokalen Zustand hält, der nicht automatisch mit anderen Threads geteilt wird.

Für den einfachsten Fall, bei dem mit einem Update Thread ein geteilter persistenter Zustand modifiziert wird, während mehrere Threads lesend auf ihn zugreifen, wird explizite Synchronisation nicht benötigt.

Ein Beispiel, welches zeigt wie beim Teilen von Zuständen Data Races entstehen wird in Abbildung 5.1 dargestellt. Hier wird durch einen Thread ein State Objekt erzeugt. Das State Objekt enthält zwei Integer Werte a und b, welche zu Beginn mit 1 und 2 initialisiert werden. Danach wird ein neuer Thread gestartet, welcher die Summe der beiden Werte des momentanen Zustandes berechnen soll. Der State wird hierbei als Objektreferenz übergeben.

Der SumCalculatorthread startet die Berechnung, in dem eine Leseoperation auf dem State Objekt ausgeführt wird und der Wert von a gelesen wird. Da keine Synchronisation vorgegeben ist, wird bei manchen Läufen des Programmes vom Betriebssystem zu diesem Zeitpunkt wieder der Hauptthread ausgeführt. Hier wird eine Update-Operation ausgeführt, bei der beide Werte geändert werden. Nach dem ersten Schritt der Änderung wird wieder der SumCalculatorthread ausgeführt. Dies ist möglich,

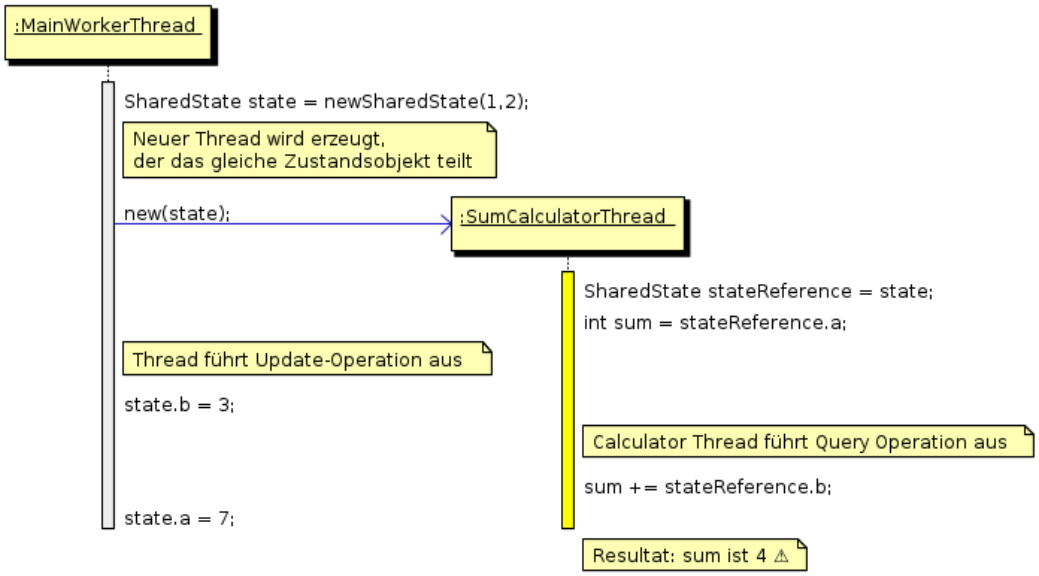


Abbildung 5.1: Data Race in geteiltem Zustand

da die Änderung keine atomare Operation ist und unterbrochen werden kann. Dieser führt nun den Rest der Berechnung aus, und erhält einen Wert als Resultat, welcher aus den Einzelwerten der Ursprungsversion sowie einer halb-modifizierten Version berechnet wurde. Dieses Problem kann durch explizite Synchronisierung gelöst werden in dem der Zugriff auf das State Objekt, während Update-Operationen durchgeführt werden, blockiert wird. Das Blockieren kann entweder für die Dauer die benötigt wird, um eine Kopie des Zustandes für den SumCalculatorthread zu erzeugen oder über die gesamte Dauer der Berechnungsoperation angewandt werden. Die Synchronisation muss jedoch explizit vom Programmierer festgelegt werden und ist anfällig für Fehler. Es muss genau darauf geachtet werden, welcher Bereich synchronisiert werden muss.

In Abbildung 5.2 wird dargestellt, wie persistente Objekte helfen diese Probleme zu beheben.

Der SumCalculatorthread erhält wie zuvor eine Referenz auf den geteilten Zustand. Da dieses Objekt nun persistent ist, können sich die Werte nicht mehr ändern. Die Berechnung kann durchgeführt werden und ist nicht von der Ausführungsreihenfolge der Threads abhängig. Der Update Thread kann dennoch Modifikationen auf dem persistenten Objekt ausführen. Er erhält aber im Unterschied zur nicht persistenten Implementierung, bei Modifikationen neue Objekte. Diese beeinflussen den Zustand mit dem vom SumCalculatorthread gearbeitet wird nicht. Auch wenn die geteilte Referenz nun auf das neue Objekt zeigt, können die Query Threads noch die alte Version des Zustandes verwenden, falls sie eine Referenz auf dieses gespeichert haben.

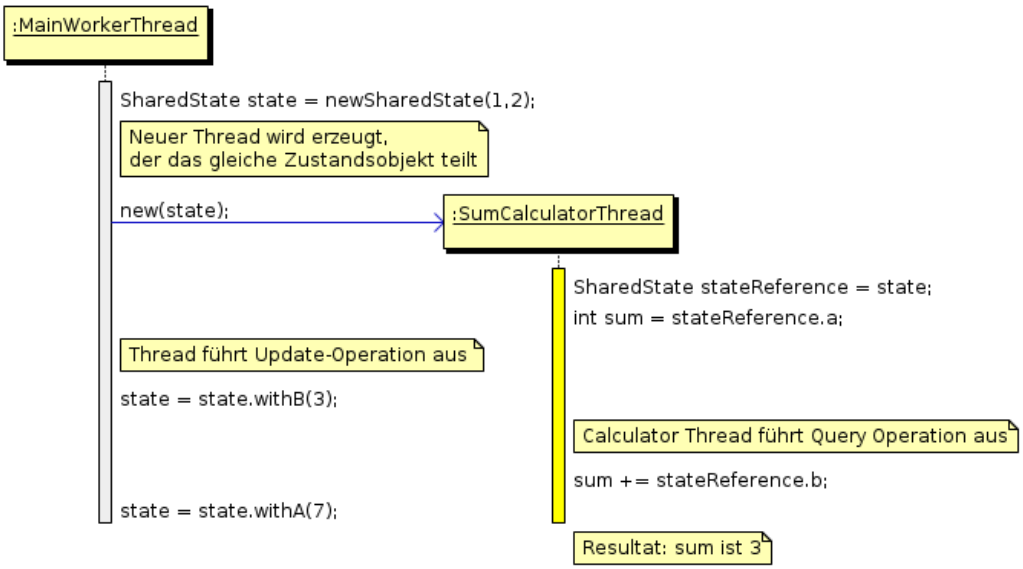


Abbildung 5.2: Geteilter Zustand mit persistenter Implementierung

6 Entwickeltes Framework zur Verwaltung von geteilten Zuständen

6.1 Problematik

Bei der Verwendung von persistenten und unveränderbaren Klassen in asynchronen Umgebungen treten Probleme auf, die die Verwendung als persistente Zustände erschweren. Die Verwendung von persistenten Datenstrukturen ermöglicht es zwar, explizite Synchronisation zu vermeiden. Allerdings werden Änderungen jeweils Threadlokal durchgeführt. Dies führt dazu, dass ein geteilter Ursprungszustand voll parallel von mehreren Threads modifiziert werden kann. Dadurch hält jeder Thread den Zustand mit dem von ihm ausgeführten Änderungen. Dies stellt aber in vielen Anwendungsfällen ein Problem dar, da die Modifikationen der einzelnen Threads wieder in einen gemeinsamen Zustand übergeführt werden müssen.

Dies ist oft bei Anwendungen nötig, bei denen unterschiedliche Threads basierend auf einem Ursprungszustand Änderungen durchführen, welche von den anderen Threads später mitverwertet werden sollen.

Bei der Zusammenführung der geänderten Zustände muss darauf geachtet werden, dass der abgeleitete neue Zustand in sich konsistent ist.

6.2 Versionsverwaltung

Ähnliche Probleme wie im vorherigen Kapitel beschrieben, treten bei der Versionsverwaltung von Texten und Sourcecode auf. Auch dort werden Daten modifiziert, welche dann wieder zusammengebracht werden müssen. In der Versionsverwaltung gibt es zwei Strategien, mit denen ermöglicht wird eine Historie der Änderungen aufzubauen und die Dateien zu verwalten. Das einfachere Verfahren wird als "Lock Modify Write"bezeichnet. Hierbei kann eine Datei nur von einem Benutzer modifiziert werden. Die Datei wird gesperrt, wenn ein Nutzer eine Modifikation durchführen will (Lock). Dieser führt die Modifikation aus (Modify) und das Versionsverwaltungssystem übernimmt die Änderung. (Write) Da nur ein Benutzer die Datei zu

einem bestimmten Zeitpunkt bearbeiten kann, werden Konflikte vermieden. Allerdings müssen Nutzer, die an einer bereits gesperrten Datei Änderungen vornehmen wollen, warten bis diese wieder freigeben wird. Dies ist analog zur Synchronisation von Prozessen zu sehen. Dort wird, falls eine Ressource nur durch einen Thread zur gleichen Zeit benutzt oder geändert werden kann, durch explizite Synchronisation die Benutzung durch einen anderen Thread verhindert.

Um dieses Problem zu lösen, ermöglichen andere Systeme ein Vorgehen nach dem Prinzip "Copy Modify Merge". Dabei können mehrere Nutzer gleichzeitig Änderungen an jeweils lokalen Kopien durchführen. Diese werden dann, falls erwünscht, vom Versionsverwaltungssystem wieder zusammengeführt. Moderne Systeme übernehmen ein Großteil des Zusammenführens, so dass nur in Konfliktfällen vom Nutzer eingegriffen werden muss.

Ein Verfahren welches von den meisten Systemen verwendet wird ist das Drei-Wege Merge Verfahren. Dabei wird aus zwei Versionen sowie ihrem gemeinsamen Vorfahren ein neuer Nachfolger abgeleitet.

Prinzipiell werden die Zeilen der Textdateien gegenüber ihrem gemeinsamen Vorfahren verglichen und Änderungen gegenüber dem Vorgänger übernommen. Bei Zusammenführungsprozessen, bei denen in beiden Nachfolgeversionen unterschiedliche Änderungen an den gleichen Zeilen vorgenommen worden sind, kommt es zu sogenannten Konflikten. Diese werden dem Nutzer angezeigt der das Zusammenführen ausführt. Er muss dann manuell die Konflikte beheben. Diese Mergekonflikte verursachen auf Grund ihrer Komplexität oft neue Fehler und sind schwer für den Benutzer zu beheben, jedoch sind sie durch das "Copy Modify Merge"Verfahren bedingt und lassen sich nicht vermeiden.

Auch beim Zusammenführen von Objekten kommt es zu diesen Problemen. Statt Zeilen von Texten zusammenzuführen, werden Objekte erzeugt, die Verweise auf die Objekte enthalten auf die in den zusammenführenden Objekte verwiesen wird. Konflikte entstehen, wenn beide zusammenführenden Objekte gegenüber dem gemeinsamen Vorgänger geänderte Referenzen enthalten. Auch hier muss dieser Konflikt behoben werden.

6.3 Zielsetzung

Für das Problem des Zusammenführens von unterschiedlich modifizierten Zuständen soll im Rahmen der Bachelorthesis untersucht werden, ob es möglich ist ein Framework in Java zu entwickeln, welches den Prozess des Zusammenführens vereinfacht und möglichst allgemein löst.

Das zusammengeführte Objekt, im nachfolgenden Resultat genannt muss mehreren Anforderungen genügen. Es muss einen validen Zustand repräsentieren. Dies bedeutet, dass die einzelnen Felder des Objektes zusammen passen müssen. Das Resultat soll außerdem die Unveränderbarkeits-Eigenschaft nutzen, um unnötiges Kopieren von Objekten zu vermeiden. Dies kann durch das Verweisen von Referenzfeldern des Resultates auf die in den zusammenzuführenden Objekten referenzierten Objekte erreicht werden.

Das Framework soll es ermöglichen, dass Anwender eigene Zustandsklassen entwickeln, welche von dem Zustandsmanager des Frameworks verwaltet werden. Die Zustände müssen als unveränderbare Klassen implementiert werden, da das Framework beim Zusammenführen Annahmen über die Unveränderbarkeit der im Zustand enthaltenen Objekte trifft. Das Framework übernimmt die Versionierung der Zustände sowie das Zusammenführen von modifizierten Zuständen.

6.4 Vorgehen

Das Verwalten von Zuständen wird über eine Verwaltungsinstanz geregelt. In ihm werden alle validen Versionen, die zu einem Zeitpunkt den aktuellen Zustand repräsentierten abgelegt. Außerdem wird in ihm der aktuelle Zustand gehalten, auf dem basierend Threads Änderungsoperationen durchführen können. Modifizierte Zustände können durch eine Methode in den StateManager zurückgeschrieben werden. Diese kapselt das Zusammenführen der Zustände, falls dies nötig ist.

Im einfachsten Fall ist der neu zu setzende Zustand ein direkter Nachfolger des aktuellen Zustands. Da sichergestellt ist, dass keine Informationen verloren gehen, kann er ohne zusammenführen direkt als aktueller Zustand gesetzt werden. Abbildung 6.1 zeigt einen solchen Fall.

Der aktuelle Zustand, sowie die bisherigen aktuellen Zustände werden in einem StateManager Objekt gehalten. Vom gemeinsamen Zustand mit der Versions-Id 1 leitet Thread 1 durch eine Änderung einen Thread-lokalen Zustand mit der Versions-Id 2 ab. Alle Zustände beinhalten Versionsobjekte, welche auf das Versionsobjekt des jeweiligen Vorgängerzustands verweist, der modifiziert wurde. Dadurch wird eine Versionskette realisiert, die benutzt werden kann um Änderungen zwischen Versionen nachzuvollziehen. Die Versionskette wird in der Abbildung mit grünen Pfeilen dargestellt.

Zur gleichen Zeit modifiziert Thread 2 den gemeinsamen Zustand und erhält dadurch einen eigenen Thread-lokalen Zustand 3. Nun wird in Thread 1 eine weitere Modifikation durchgeführt. Diese soll nun wieder als gemeinsamer Zustand gesetzt werden.

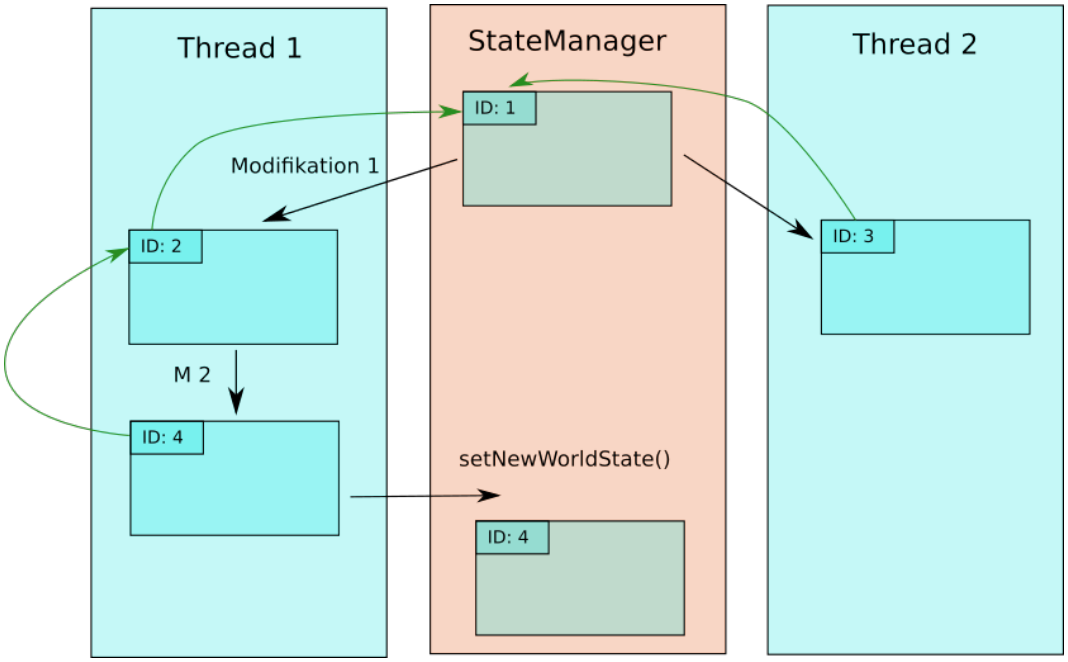


Abbildung 6.1: Statemerger mit direktem Nachfolger

Von Thread 1 wird die Methode setNewWorldState des StateManagers aufgerufen und der Zustand 4 übergeben.

Durch die Versionierung kann im StateManager herausgefunden werden, ob das neue Objekt von dem momentan gültigen Zustand abgeleitet wurde. Da Objekt 4 ein direkter Nachfolger von Objekt 1 ist, und die Zustandsobjekte persistent sind, kann durch das Setzen des Objekts mit der Id 4 keine Änderung verloren gehen, welche bereits in den Zustand im Statemanager eingearbeitet wurde. Deshalb kann das modifizierte Objekt direkt als neues Objekt gesetzt werden. Die Änderung die von Thread 2 in den Zustand 3 eingearbeitet wurde hat keinerlei Einfluss auf den neuen Zustand und beeinträchtigt auch nicht das Setzen des neuen, gemeinsamen Zustandes.

Schwieriger ist es, falls der neu zu setzende Zustand kein direkter Nachfolger des aktuell im StateManager enthaltenen Zustands ist. Eine solche Situation wird in Abbildung 6.2 dargestellt. Hier leitet Thread 1 einen neuen Zustand ab, welcher den String t1 gegenüber Zustand 1 verändert. Der neue Zustand wird bei Punkt 1 im StateManager gesetzt, dies kann wie im vorherigen Beispiel ohne Zusammenführung durchgeführt werden. Danach will Thread 2, der auch einen Zustand von Zustand 1 abgeleitet hat, diesen als aktuellen Zustand des StateManager setzen.

Da Zustand 3 kein direkter Nachfolger vom momentanen Zustand 2 ist, muss nun ein Zusammenführungsverfahren durchgeführt werden. Hierbei muss ein neuer Zu-

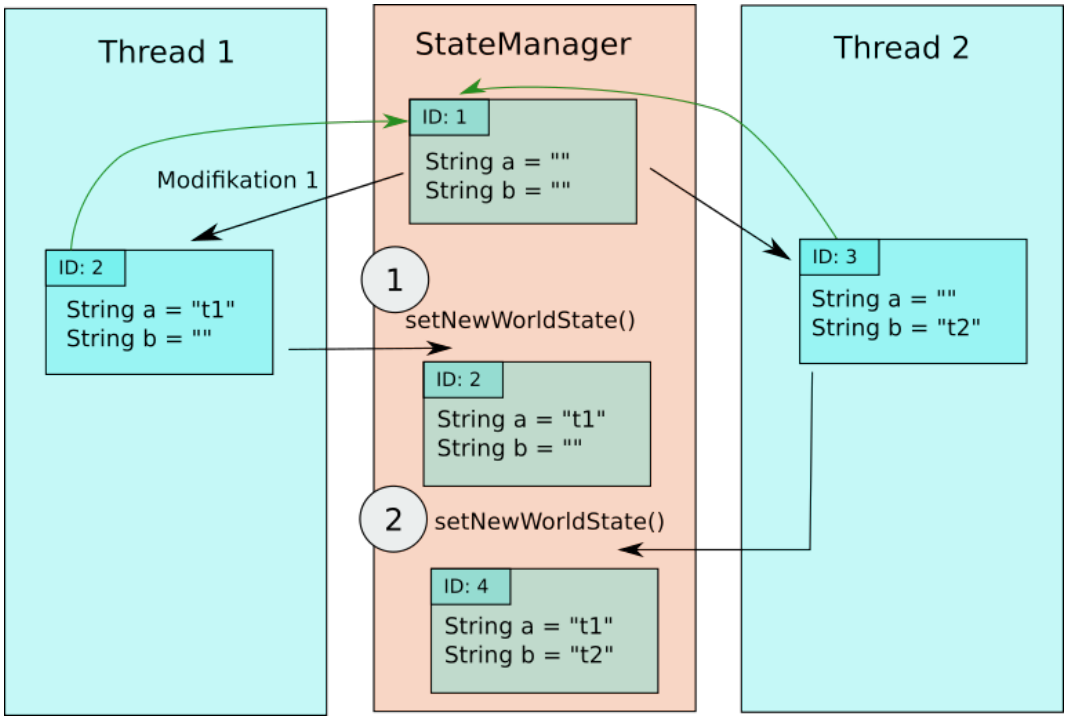


Abbildung 6.2: Statemerger mit Zusammenführen

stand 4 erzeugt werden, welcher sowohl die Änderungen von Zustand 2 als auch die Änderungen von Zustand 3 gegenüber dem Ursprungszustand 1 beinhaltet.

6.5 Umsetzung als Java Framework

Die Überlegungen über das Zusammenführen von Zuständen sollen in einer Implementierung in Java umgesetzt werden. Mit ihr soll die Machbarkeit untersucht werden, sowie Messungen durchgeführt werden, um diesen Ansatz mit anderen wie der expliziten Synchronisation zu vergleichen.

6.5.1 Konzeption

Zunächst soll auf die Ergebnisse der grundlegenden Überlegungen zum Umsetzen eines Framworks zur Lösung des Problemes von geteilten Zuständen eingegangen werden. Danach werden spezifische Details der Implementierung vorgestellt.

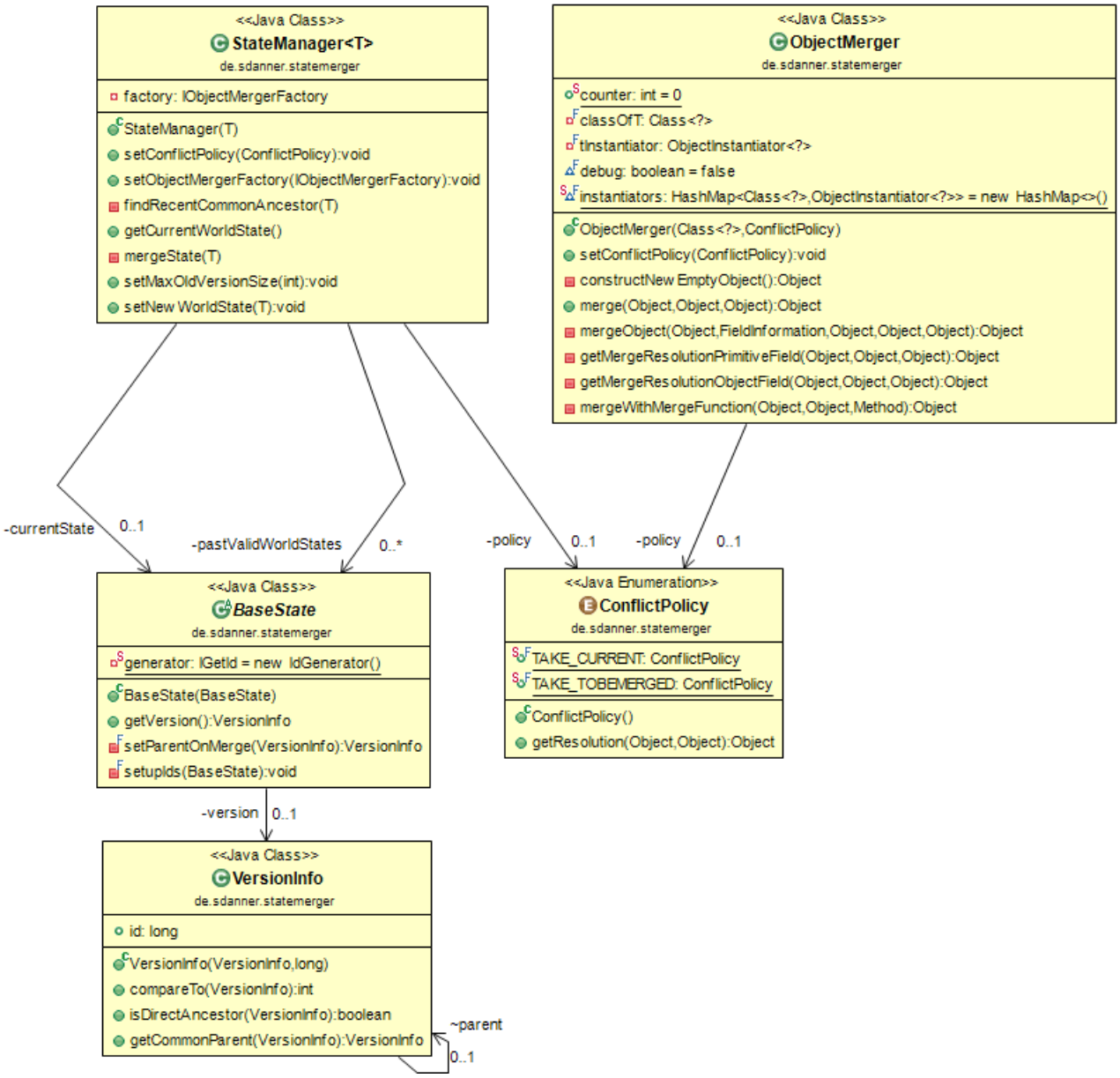


Abbildung 6.3: UML Diagramm des entwickelten Frameworks

6.5.1.1 Architektur

In Abbildung 6.3 wird die Architektur des entwickelten Frameworks zur Übersicht als UML Diagramm dargestellt. Auf die einzelnen Komponenten wird im Folgenden eingegangen.

6.5.1.2 BaseState

Um das Framework zu benutzten, muss eine Klasse implementiert werden die von der abstrakten Klasse BaseState erbt. In ihr ist die Verwaltung der Versionsinformationen gekapselt, welche beim Neuerzeugen eines Objektes die Verlinkung zur Versionsinformation des Ursprungsobjektes vornimmt. Dies geschieht im Konstruktor der BaseState Klasse, welcher bei jeder Modifikation eines Zustandobjektes aufgerufen wird. Der Aufruf ist durch die zwingende Unveränderbarkeit der vom Anwender implementierten Klasse garantiert. Dadurch wird bei jeder Änderungsoperation ein neues Objekt erzeugt. Somit kann sichergestellt werden, dass alle Zustände eine korrekte Versions-Information beinhalten, die auf den Vorgängerzustand verweist.

6.5.1.3 StateManager

Die StateManager Klasse ist generisch über Klassen die von BaseState erben. Ein StateManager enthält den momentanen Zustand, sowie eine Sammlung bisheriger momentaner Zustände. Sie ist der Kern des Frameworks, welcher von den einzelnen Threads benutzt wird, um den momentanen Zustand abzufragen. Die einzelnen Threads einer Anwendung benötigen eine Referenz auf den StateManager, welcher den jeweiligen Zustand der von den Threads geteilt werden soll verwaltet. Falls benötigt, können in einer Anwendung mehrere StateManager angelegt werden, die jeweils andere Zustandsklassen verwalten.

Über eine Methode kann ein Thread eine Objektreferenz zum aktuellen Zustand bekommen. Mit diesem Zustand arbeitet der Thread und führt eventuell Update-Operationen auf ihm aus. Da die Zustände unveränderbare Objekte sind, werden bei Update-Operationen neue Zustände erzeugt, welche nur dem aktuellen Thread bekannt sind. Wenn die durchgeführten Änderungen den anderen Threads verfügbar gemacht werden sollen, kann auf dem StateManager die Methode setNewState aufgerufen werden, welche den neuen Zustand in den StateManager einarbeitet. Hierbei wird überprüft ob der neue Zustand den alten aktuellen Zustand ersetzen kann, oder ob der neu einzupflegende Zustand mit dem aktuellen Zustand zusammengeführt werden muss.
Die bisherigen im StateManager gesetzten Zustände werden in ihm gespeichert. Dies ist nötig, damit später beim Untersuchen neu einzuarbeitender Zustände gegenüber dem gemeinsamen Vorfahren, diese noch existieren. Es kann über eine Methode eine maximale Anzahl der zu speichernden Zustände festgelegt werden. Dies begrenzt den Speicherverbrauch bei lang laufenden Prozessen. Dadurch dass die VersionsInfo Objekte nur auf andere VersionsInfo Objekte zeigen und nicht auf die zugehörenden Zustände, ist es möglich dass nicht alle bisherigen Zustände im Speicher behalten

werden müssen, während die Versionshistorie besteht. Die Speicher für nicht mehr vorgehaltene Zustände kann also durch den Garbage Collector freigegeben werden. Wenn der gemeinsame Vorfahre nicht vorgehalten wird, kann der neu einzupflegende Zustand nicht verarbeitet werden. Dies ist jedoch nur der Fall, wenn ein Thread Modifikationen auf einem zu alten Zustand ausgeführt hat. Das könnte auftreten, wenn ein einzelner Thread viel langsamer als ein anderer Thread neue Zustände erstellt und diese in den StateManager einarbeitet. Durch Anpassung der Anzahl vorgehaltener Zustände kann das aber anwendungsspezifisch vermieden werden. Modifikationen sollten durch die Threads also, wenn möglich, auf einem aktuellen geteiltem Zustand ausgeführt werden.

6.5.1.4 ObjectMerger

Die ObjectMerger Klasse erzeugt ein neues Objekt aus einem neu einzupflegenden Objekt, einem aktuellen Objekt, sowie einem Objekt das ein gemeinsamer Vorgänger der beiden ist. Das resultierende Objekt muss sowohl alle Änderungen des aktuellen, als auch des neu einzupflegenden Objektes gegenüber dem gemeinsamen Vorgänger enthalten.

Der ObjectMerger trifft die Annahme, dass die zusammenführenden Objekte unveränderbar sind. Wäre dies nicht der Fall, könnte eine Änderung an einem der zusammenführenden Objekte, welche nach dem Zusammenführen ausgeführt wird, eine Änderung am Resultat bewirken. Dies ist nicht erwünscht, da unter anderem dadurch das Ermitteln der Unterschiede zum Vorgänger unmöglich wird. Im Framework wird der ObjectMerger benutzt, wenn ein neuer Zustand als gemeinsamer Zustand im StateManager gesetzt werden soll. Falls der neue Zustand kein direkter Nachfolger des momentanen Zustands ist, müssen die Änderungen mit dem aktuellen Zustand zusammengeführt werden. Dafür wird über die Versionskette des neuen Zustandes und des aktuellen Zustandes der gemeinsame Vorgänger ermittelt. Der neue Zustand, der gemeinsame Vorfahre und der aktuelle Zustand werden dann dem Objektmerger übergeben, der ein neues Objekt erzeugt indem alle Änderungen enthalten sind. Dies geschieht durch das Ermitteln der Unterschiede zwischen dem gemeinsamen Vorgänger und den beiden anderen Zuständen. Änderungen werden in das neu erzeugte Objekt übernommen, welches dann durch den Statemanager als neuer aktueller Zustand gesetzt wird.

Aufgrund der Resultate der Vergleiche wird die Entscheidung getroffen, welcher der Werte in das Resultat übernommen werden soll.

Die Entscheidung wird nach Tabelle 6.1 getroffen.

Änderung von Objekt A gegenüber Vorfahren	Änderung von Objekt B gegenüber Vorfahren	Entscheidung
true	false	A
false	true	B
false	false	Vorfahren
true	true	Rekursion oder Policy

Tabelle 6.1: Merge-Entscheidung

Im Falle, dass ein Feld beider Zustände gegenüber dem Vorfahren verändert ist, wird rekursiv in die im Konflikt stehenden Objekte abgestiegen und versucht die von ihnen referenzierte Objekte zusammengeführt. Wenn ein Punkt erreicht wird, bei dem nicht zusammengeführt werden kann, wird anhand einer Richtlinie entschieden. Außerdem können Funktionen die bestimmte Felder zusammenführen angegeben werden.

6.5.1.5 VersionInfo

Ein Objekt der VersionsInfo Klasse ist in jedem von BaseState abgeleiteten Objekt enthalten. In ihr wird eine eindeutige Identifikationsnummer gekapselt, sowie eine Referenz zu dem VersionsInfo Objekt des Elternzustands. Dies ermöglicht es von jedem Zustand die Versionshistorie, durch verfolgen der Referenzkette, zu ermitteln. Die Versionshistorie kann somit dazu benutzt werden, um die Zustände auf Änderungen zu ihren Vorgängern zu vergleichen.

6.5.2 Implementierung

Im Folgenden Kapitel sollen die konkreten Implementierungen der benötigten Komponenten vorgestellt werden und in den Komponenten zu lösende Probleme aufgezeigt werden.

6.5.2.1 Erzeugung neuer Objekte

Für die Implementierung des Objektmergers mussten zunächst einige grundsätzliche Probleme gelöst werden. Zunächst muss eine Komponente implementiert werden, mit der Objekte ohne den Aufruf ihrer Konstruktoren erzeugt werden können. Dies ist nötig, da der Code des Frameworks die Zusammenführungsoperation auf beliebigen, von einer StateBase Klasse abgeleiteten Objekten durchführen soll. Das Aufrufen des Konstruktors für den Fall, dass ein neues Objekt erzeugt werden muss ist allerdings nicht möglich. Der Code der ein neues Objekt aus bisherigen Objekten erzeugen

will, kann nicht ableiten welcher Konstruktor mit welchen Argumenten aufgerufen werden muss.

Wenn eine leere Instanz der Klasse des neu zu erzeugenden Objektes konstruiert werden kann, können die Felder durch Java Reflection gesetzt werden. Eine Möglichkeit dies zu erreichen wäre zu fordern, dass alle Klassen die in den zusammenzuführenden Objekten enthalten sind einen leeren Konstruktor implementieren müssen. Dieser könnte dann über Reflection ausgeführt werden, und die Felder des resultierenden Objektes mit Reflection gesetzt werden. Da dies jedoch eine Einschränkung für die Verwendung des Frameworks darstellen würde, wurde ein anderer Lösungsweg gewählt.

6.5.2.2 Objenesis

Die unter einer Apache Lizenz stehende Objenesis Bibliothek ermöglicht das erstellen von Objekten ohne Aufruf des Konstruktors. Hierfür benutzt die Bibliothek Java VM spezifische Implementierungen, welche es ermöglichen leere Objekte des richtigen Typs zu erzeugen. Objenesis stellt hierfür Backends für die Oracle Hotspot VM, OpenJDK, Androids Dalvik VM, JRockit und GCJ bereit. Als Benutzer der Bibliothek ist man somit plattformunabhängig und muss sich nicht um die Details der Umsetzung kümmern.

Listing 6.1: Objenesis

```
1 Objenesis objenesis = new ObjenesisStd();
2 ObjectInstantiator toCreateInstantiator = objenesis.getInstantiatorOf(ToCreate.class);
3
4 ToCreate a = (ToCreate)toCreateInstantiator.newInstance();
```

In Listing 6.1 wird gezeigt wie Objenesis für die Erzeugung einer Instanz von ToCreate verwendet werden kann. Optional kann das Objenesisobjekt mit einer Erzeugungsstrategie instantiiert werden, was jedoch nur Sinn macht, wenn festgelegt ist auf welcher Java VM der Code ausgeführt wird, da diese plattformabhängig sind. Wird keine Strategie explizit gewählt wird automatisch die beste Strategie für die momentane Plattform ermittelt und verwendet. Mit getInstantiatorOf wird ein ObjectInstantiator erzeugt und zurückgegeben. Die ObjectInstantiatoren werden im Objenesis Objekt im Normallfall zwischengespeichert.

Nach dem Erzeugen eines Instantiators für eine Klasse, kann dieser beliebig viele neue Objekte der Klasse erzeugen, die keine Felder gesetzt haben.

Dadurch sind die erzeugten Objekte nur für sehr spezielle Einsatzbereiche benutzbar. Der Code von Objenesis entstand aus den Anforderungen des Mockingframeworks EasyMock, sowie zweier Serialisierungsbibliotheken.

Für die Implementierung des Frameworks können die von Objenesis erzeugten Objekte verwendet werden, um per Reflection die Referenz-Felder auf die Objekte die von anderen Versionen referenziert werden, zeigen zu lassen.

6.5.2.3 Reflection

Java bietet mit der Reflection API eine Möglichkeit zur Laufzeit auf Klassen, Interfaces, Felder und andere Elemente zu untersuchen und verwenden. Durch die Verwendung von Reflection im Framework können Objekte zusammengeführt werden, deren interner Aufbau dem Framework nicht bekannt ist. Reflection ermöglicht die Member eines Objektes zu ermitteln, diese zu lesen und zu schreiben, selbst wenn diese im Code durch Zugriffsmodifikatoren gesichert sind.

6.5.2.4 Objectmerger

Für das Framework wird angenommen, dass die vom Anwender entwickelten Zustandsobjekte unveränderbar implementiert sind. Durch diese Annahme können Referenzfelder des Resultates auf die Objekte zeigen, auf die in den zu zusammenführenden Objekten verwiesen wird. Die Speicherverwaltung in Java garantiert, dass Objekte auf die Verweise existieren nicht gelöscht werden. Dadurch kommt es bei Verweisen auf von anderen Objekten referenzierte Objekte nicht zu Problemen, auch wenn das erste referenzierte Objekt gelöscht wird. In anderen Sprachen mit manueller Speicherverwaltung wie C++ kann diese Annahme nicht ohne weiteres getroffen werden.

Um ein neues Objekt zu erzeugen, wird durch Objenesis ein neues, uninitialisiertes Objekt des Typs der zu zusammenführenden Objekte erstellt. Danach werden durch Reflection die Felder des Objektes und Informationen zu ihnen rekursiv ermittelt. Da die Felder eines Objektes zur Laufzeit sich nicht ändern, werden die Feldinformationen im ObjectMerger zwischengespeichert. Dies verbessert die Performance, da der ObjectMerger in den meisten Fällen während der Laufzeit einer Anwendung immer für die gleichen Typen verwendet wird. Die Rekursivität bezieht sich hierbei auf die Vererbungshierarchie, sodass auch Felder von Superklassen ermittelt werden.

Danach wird durch die Liste der Felder iteriert. Von jedem Feld wird der Wert der drei verwendeten Objekte ermittelt. Durch die Java Reflection API werden hier Objektreferenzen zurückgegeben. Aufgrund der drei Werte kann nun ermittelt werden ob sich keine, eine Version oder beide Versionen gegenüber des gemeinsamen Vorfahren geändert hat.

Bei der Behandlung der Referenzen muss zwischen den beiden in Java vorhandenen Typ-Kategorien unterschieden werden. Die erste Kategorie sind die Referenz Typen. Hierzu zählen in Java Klassen, Enumerationen, Arrays und Interfaces. Die andere Kategorie bilden die sogenannten primitiven Typen. Java besitzt folgende primitive Typen: boolean, byte, short, int, long, char, float und double.

Wenn ein Feld einen Referenz-Typ hat, wird die Ermittlung von Änderungen durch einen einfachen Vergleich der Referenzen durchgeführt. Ist die Referenz gegenüber des Vorfahrzustandes geändert, ist eine Änderung durchgeführt worden. Wenn der Typ des Feld primitiv ist, kann nicht die Referenz verglichen werden, es muss zunächst der Wert ermittelt werden. Dies ist nötig da die Reflection API in Java Werte von Feldern primitiver Typen automatisch in Objekte umformt. Das sogenannte Boxing ist in diesem Fall jedoch nicht erwünscht, da für den gleichen primitiven Wert unterschiedliche Objekte erzeugt werden. Deshalb muss zunächst der konkrete Datentyp ermittelt werden, und durch einen Cast auf diesen die enthaltenen Werte verglichen werden.

Durch die Ermittlung von Änderungen kann nach Tabelle 6.1 entschieden werden, welches Objekt vom Resultat referenziert werden soll.

Falls beide Objekte gegenüber der Ursprungsversion verändert sind, werden die kollidierenden Objekte rekursiv durch einen neu erzeugten Objektmerger zusammengeführt. Dabei muss aber beachtet werden, ob die Klasse Felder besitzt, welche unter bestimmten Bedingungen von einander abhängen. Dies wird im Kapitel Einschränkungen erläutert. Aufgrund dieser Einschränkung werden nur Objekte, deren Klasse eine vorgegebene Annotation besitzen, rekursiv zusammengeführt.

Hierbei werden wieder die Unterschiede gegenüber dem Ursprungsobjekt ermittelt und ein neues Objekt basierend auf den Resultaten der Vergleiche der Felder erzeugt. Dies wird solange durchgeführt, bis eine Ebene erreicht wird, indem kein Konflikt mehr besteht, oder ein Objekt erreicht wird, welches nicht weiter zusammengeführt werden kann. Dies ist bei primitiven Datentypen der Fall, da sie keine anderen Objekte mehr enthalten. Auch bei Arrays wird abgebrochen, da nicht domänenunabhängig entschieden werden kann, welche Elemente welches Arrays in dem neu erzeugten Objekt enthalten sein sollen. Siehe Kapitel Einschränkungen.

In diesen Fällen muss eine Entscheidung getroffen werden, welche Änderung verworfen bzw. welche Änderung benutzt werden soll. Im Gegensatz zum Zusammenführen bei Versionsverwaltungssystemen ist es nicht praktikabel die Konflikte durch einen Menschen zu lösen. Zum einen würde die manuelle Konfliktbehebung zu lange dauern, zum anderen müsste der Konflikt so dargestellt werden, dass der die Maschine bedienende Mensch ihn sinnvoll beheben kann. Bei Konflikten welche aus widersprüchlichen Code-Zeilen bestehen ist dies aufgrund des Anwenders (Entwickler)

und des Kontextes (umgebender Code) sowie der lockereren zeitlichen Beschränkungen einfacher auszuführen als bei Konflikten bei unterschiedlichen Werten von Referenzfeldern.

Deshalb muss es im Framework eine Möglichkeit geben, solche Konflikte durch ein festgelegtes Verhalten zu beheben. Dies wird durch eine Richtlinie umgesetzt, welche im Objektmerger berücksichtigt wird. Hierbei gibt es zwei Möglichkeiten, welche in Richtlinien abgebildet werden. Im Framework sind diese wie in Tabelle 6.2 dargestellt enthalten.

Policy	Änderung Objekt A gegenüber Vorfahren	Änderung Objekt B gegenüber Vorfahren	Ergebnis
TAKE_CURRENT	true	true	A
TAKE_NEW	true	true	B

Tabelle 6.2: Merge-Entscheidung bei Konflikt

Zusätzlich zu dieser einfachen Konfliktlösung mit Hilfe der Richtlinien, können explizit Methoden angegeben werden, welche bei Konflikten der spezifizierten Felder ausgeführt werden. Die Methoden müssen hierbei ein Objekt des gleichen Typs entgegennehmen, wie des Feldes bei dem der Konflikt aufgetreten ist. Außerdem muss ein Objekt des gleichen Typs zurückgegeben werden. Außer diesen Vorgaben gibt es keine weiteren Beschränkungen. Es kann beliebiger Code ausgeführt werden. Dies ermöglicht große Flexibilität für den Anwender des Frameworks, da er frei entscheiden kann was im Konfliktfall passiert.

Listing 6.2: Beispiel explizite Konfliktbehebungs-Methode

```
1  class classWithList {
2    @MergeAnnotation(method = "mergeList")
3    private List<Integer> list;
4
5    public List<Integer> mergeList(List<Integer> list2) {
6      List<Integer> l1 = new LinkedList<>(list2);
7      l1.addAll(list);
8      return l1;
9    }
10 }
```

Die Methoden können durch Annotationen im Sourcecode angegeben werden. In 6.2 wird ein minimales Beispiel gezeigt, bei dem im Konfliktfall des Feldes list die Methode mergeList() der Klasse aufgerufen wird. Wenn eine Methode explizit angegeben ist, wird diese höher priorisiert als die Entscheidung nach Richtlinie. Durch die Annotation kann für jedes Feld eine beliebige Methode angegeben werden, welche aus der aktuellen Instanz und einer zu mergenden Instanz ein neues Objekt ableitet. Die Methode kann auch als private Methode implementiert werden, falls nicht

erwünscht wird die Methode für andere Klassen außerhalb des Frameworks nutzbar zu machen. Das Framework selbst ignoriert Zugriffsmodifikatoren.

6.5.3 Einschränkungen der Implementierung

Aufgrund von technischen und prinzipiellen Gegebenheiten bestehen im Framework verschiedene Einschränkungen, welche die Verwendung erschweren.

6.5.3.1 Arrays

Arrays können nicht durch Objenesis erzeugt werden. Dadurch ist es nicht möglich generisch neue Array Objekte anzulegen. Dies wird im Falle des Objektmergers jedoch nicht zwingend benötigt, da es bei Arrays nicht Domänen-unabhängig möglich ist zu entscheiden, welche Elemente in das neu zu erzeugende Objekt übernommen werden sollen. Um dieses Problem zu lösen, kann die Möglichkeit der expliziten Merge-Funktion, die in 6.5.1.4 beschrieben ist, genutzt werden.

6.5.3.2 Fall-abhängige Konstruktoren

In speziellen Fällen kann es beim automatisierten Zusammenführen zu Resultaten kommen, welche keine validen Objekte darstellen. Die Resultate haben hierbei einen inneren Zustand, welcher nicht durch normale Modifikationen des Objektes mit seinem bereitgestellten Methoden erreicht werden kann. Dies wird durch fallabhängige Zuweisung von Werten an Member des Objektes verursacht. Da Zuweisungen an Membern bei unveränderbaren Objekten nur im Konstruktor ausgeführt werden können, müssen nur die Konstruktoren der Klasse betrachtet werden.

Listing 6.3: Beispiel Zusammenführungsfehler bei abhängigen Membern

```
1  package de.sdanner.statemerger.test;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  import de.sdanner.statemerger.ConflictPolicy;
8  import de.sdanner.statemerger.ObjectMerger;
9
10 public class ThesisDemoNotWorking {
11
12   class ObjectHolder<T> {
13
14     public final Object primaryStorage;
15     public final Object overflowStorage;
16     private final int containedCount;
17
```

```
18    public ObjectHolder<T> store(T val) {
19      if(val==null)
20      {
21        return this;
22      }
23      // room in primaryStorage ?
24      if (containedCount < 1) {
25        return new ObjectHolder<T>(containedCount + 1, val,
26            overflowStorage);
27      }
28      // full primaryStorage, use overflowStorage
29      return new ObjectHolder<>(containedCount + 1, primaryStorage, val);
30    }
31
32    public int count() {
33      int count = 0;
34      if (primaryStorage != null)
35        count++;
36      if (overflowStorage != null)
37        count++;
38      return count;
39
40    }
41
42    private ObjectHolder(int containedObjects, Object primaryStorage,
43        Object overflowStorage) {
44      this.containedCount = containedObjects;
45      this.overflowStorage = overflowStorage;
46      this.primaryStorage = primaryStorage;
47    }
48
49    public ObjectHolder() {
50      overflowStorage = null;
51      primaryStorage = null;
52      containedCount = 0;
53    }
54  }
55
56  @Test
57  public void showMergeDependendDoesntWork() throws IllegalArgumentException,
58      IllegalAccessException {
59    ObjectHolder<Integer> initial = new ObjectHolder<Integer>();
60
61    ObjectHolder<Integer> toMerge = initial.store(2);
62    ObjectHolder<Integer> current = initial.store(3).store(4);
63
64    ObjectMerger merger = new ObjectMerger(initial.getClass(),
65        ConflictPolicy.TAKE_TOBEMERGED);
66
67    ObjectHolder<Integer> result = (ObjectHolder<Integer>) merger.merge(
68        current, toMerge, initial);
69
70    assertEquals(result.count(), result.containedCount);
71    // result.count() ist 2, result.containedCount ist 1
72  }
73
74}
```

In Listing 6.3 wurde eine Klasse implementiert, die das Problem aufzeigen kann. Ein Objekt der Klasse kann zwei Objekte halten. Die Klasse ist unveränderbar implementiert. Das unsichere Verwenden der Objekte ohne sie zu kopieren kann ignoriert werden, es ist hier nicht relevant.

Im gezeigten Unit Test wird der Fehlerfall aufgezeigt. Es werden drei Objekte des Objektholders erzeugt. Das initiale Objekt beinhaltet keine Objekte und der containedCount ist somit 0. Im toMerge Objekt wird ein Objekt abgelegt, der containedCount ist 1.

Der containedCount müsste nicht mitgezählt werden, es könnte auch durch einen Vergleich der Referenzen mit null gezählt werden, auf wieviele Objekte gezeigt wird. Dies ist in der Methode count implementiert. Bei Objekten die durch Aufruf des Konstruktors sowie der store Methode erstellt werden, kann durch einen Aufruf von object.cout() immer nur der Wert von result.containedCount zurückgegeben werden. Es gibt keine Möglichkeit als Nutzer der Klasse eine Instanz zu erzeugen, in der diese Werte nicht übereinstimmen.

Im Test wird allerdings durch die Verwendung des ObjectMergers eine nicht valide Instanz erzeugt. Um nachvollziehen zu können, warum der ObjectMerger invalide Objekte erzeugt, müssen die beinhalteten Werte betrachtet werden.

initial hält nach dem Anlegen die folgenden Werte:

containedCount	0
overflowStorage	null
primaryStorage	null

toMerge hält nach dem Anlegen die folgenden Werte:

containedCount	1
overflowStorage	null
primaryStorage	Object

current hält nach dem Anlegen die folgenden Werte:

containedCount	2
overflowStorage	Object
primaryStorage	Object

Der ObjectMerger wird mit den drei Werten aufgerufen. Hierbei wird die ConflictPolicy.TAKE_TOBEMERGED benutzt. Sie legt fest, dass bei Konflikten von Feldern,

dass heißt das beide zusammenbringende Objekte gegenüber dem initial Objekt Änderungen haben, das Feld des toMerge Objekts übernommen wird.

Beim Zusammenführen werden nun die Felder der Werte miteinander verglichen. Beim Werte von containedCount wird für das Resultat 1 gewählt, da zwar beide Objekte Änderungen gegenüber dem initial Objekt haben, aber die Merge Policy greift. Bei overflowStorage wird der Wert von current gewählt, da hier toMerge null enthält, sich also nicht gegenüber initial geändert hat. Bei primaryStorage wird wieder der Wert von toMerge gewählt, da hier wieder beide Objekte Änderungen beinhalten.

Das Resultat enthält also folgende Werte:

containedCount	1
overflowStorage	Object von current
primaryStorage	Object von toMerge

Hier würde ein Aufruf der count-Methode 2 zurückliefern, während containedCount 1 beinhaltet. Da im Code einer Klasse davon ausgegangen werden muss, dass Objekte nicht von außen manipuliert werden, ist es valid eine Methode wie count() zu implementieren. Ohne die Annahme das Objekte nicht von außen inkonsistent gemacht werden, wäre sinnvolles programmieren nicht möglich. Der Objektmerger zerstört aber durch den Zusammenführungsvorgang die innere Konsistenz der Klasse. Dies stellt ein großes Problem dar, da durch solche Inkonsistenzen undefiniertes Verhalten im laufenden Code entsteht. Der Vorteil von Vermeidung invalider Zustände bei der Verwendung von unveränderbaren Objekten wird durch den Objectmerger also zerstört, es entstehen Probleme die sehr schwer nachvollziehen zu sind.

Diese Einschränkung erschwert die Verwendung des Objektmergers sehr. Bei allen unveränderbaren Klassen, bei denen bestimmte Member nur bedingt zugewiesen werden, kann dieses Problem entstehen. Gerade bei komplexen Klassen, mit mehreren oder komplexen Konstruktoren, ist nicht ohne weiteres festzustellen, welche Klassen sicher zusammengeführt werden können. In Anwendungen wie der implementierten Beispieldomäne kann der Ansatz des Zusammenführens dennoch verwendet werden, da aufgrund der geringen Komplexität einfach überprüft werden kann, dass keine bedingten Zuweisungen von Feldern durchgeführt werden.

6.5.3.3 Komplett neue Zustände

Alle Zustände die als Modifikationen vom Zustandsmanager eingepflegt werden sollen, müssen von einem bisherigen Zustand abgeleitete Zustände sein. Das heißt, es kann kein komplett neuer Zustand während der Laufzeit des Programmes gesetzt

werden. Dies ist eine Einschränkung der Implementierung, es wäre möglich eine Reset-Methode für den Zustandsmanager zu implementieren, welche die enthaltenen bisherigen Versionen löscht, sowie nach dem zurücksetzen Versionen ignoriert die noch auf den alten Zuständen basieren.

6.6 Evaluation

Zur Evaluation des Frameworks wurde eine einfache Testdomäne entwickelt, mit der Messungen zur Performanz und zur Praktibilität des Ansatzes durchgeführt werden können. Hierbei soll ein asynchrones Modell simuliert werden, welches einen gemeinsamen Zustand besitzt, der von mehreren Threads gleichzeitig bearbeitet wird. Dabei sollen die Modifikationen wieder in einen gemeinsamen Zustand überführt werden. Es soll getestet werden, ob die Methode des Zusammenführens der StateObjekte gegenüber der expliziten Synchronisation Vorteile bringt und welche Parameter in welcher Form diese Ergebnisse beeinflussen.

6.6.1 Implementierung Testdomäne

Da die Laufzeit des Objektzusammenführens durch die Rekursivität von der Tiefe des zu zusammenführenden Objektes abhängt, bildet eine persistente Datenstruktur, von der Instanzen mit beliebiger Tiefe erzeugt werden kann, die Grundstruktur. Außerdem können die Änderungsoperationen mit der Tiefe der Änderung in der Struktur und auf die zu verändernden Felder parametrisiert werden.

Listing 6.4: Test Objekt

```
@MergeIndependentMembers
public class PersistentTreeStructure {

  final public PersistentTreeStructure one;
  final public PersistentTreeStructure two;
  ...

  public PersistentTreeStructure(int level) {
    this.level = level;
    int nextLevel = level - 1;
    if (level > 1) {
      one = new PersistentTreeStructure(nextLevel);
      two = new PersistentTreeStructure(nextLevel);
      ...
    }
    else {
      one = null;
      two = null;
      ...
    }
```

```
21   }
22   /***
23    *
24    * @param level Die Tiefe in der ein Objekt ersetzt werde soll
25    * @param selectors Selektorenlist, welche zu dem ersetzenden Objekt fuehrt
26    * @return Ein neues Objekt mit der angegebenen Modifikation
27    */
28    public PersistentTreeStructure replaceObjectAt(int level, List<Integer> selectors) {
29     ...
30    }
31 }
```

Die grobe Struktur wird in 6.4 aufgezeigt. Eine wichtige Eigenschaft der Klasse ist ihre Persistenz. Änderungen können durch die Methode replaceObjectAt durchgeführt werden. Die Methode ersetzt eines der Member Objekte mit einem neu konstruiertem Objekt, welches die gleiche Tiefe besitzt des ersetzten. Durch die Angabe von Selektoren kann gesteuert werden welches Objekt ersetzt werden soll. Die Selektoren sind eine Liste von Indizes, mit denen durch den Baum navigiert wird.

Ein Aufruf von testObject.replaceObjectAt(3, (1,2,1)) würde ein neues Objekt zurückliefern, indem testObject.one.two.one durch ein neu erzeugtes Objekt ersetzt wurde.

Um das Framework zu testen, existiert ein von der Klasse BaseState abgeleitete Zustandsklasse. In der PersistentTreeStructureState Klasse ist eine PersistentTreeStructure enthalten. Der State enthält eine Methode, mit der die replaceObjectAt Methode der enthaltenen PersistentTreeStructure aufgerufen werden kann.

Als Testprogramm wurde ein Hauptprogramm entwickelt, welches einen StateManager anlegt, welcher einen gemeinsamen geteilten PersistentTreeStructureState verwaltet.

Dieser StateManager wird von gestarteten Threads benutzt, die dann Änderungsoperationen auf dem enthaltenen State ausführen und die modifizierten States zurückschreiben. Beim Zurückschreiben führt das Framework den neu zu schreibenden Zustand mit dem aktuellen Zustand zusammen.

Durch verschiedene Parametrisierung der Threads können Messungen bezüglich des Verhaltens des Frameworks durchgeführt werden.

Die Threads können mit folgenden Parametern angelegt werden:

- Anzahl der durchzuführenden Modifikationen
- Tiefe der Modifikation
- Anzahl der abzuarbeitenden Arbeits-Operationen

Die Anzahl der durchzuführenden Modifikationen gibt an, wie oft Zustandsänderungen und das zurückschreiben in den StateManager von jedem Thread durchgeführt werden müssen, bevor er sich beendet. Die Tiefe der Modifikation gibt an, in welcher Tiefe die PersistentTreeStructure verändert wird.

Um reale Prozesse simulieren zu können, ist es nötig die Modifikationszeiten der einzelnen Threads erhöhen zu können und unabhängig von der Tiefe des PersistentTreeStructure zu machen. Die benötigte Zeit der Modifikation könnte zwar auch über die Tiefe der Modifikation der PersistentTreeStructure festgelegt werden, allerdings würde dadurch auch der Zusammenführungsprozess beeinflusst. Da diese Korrelation bei realen Anwendungen nicht unbedingt besteht, muss ein von dem zu zusammenführenden Zustand unabhängige Operation benutzt werden, um die benötigte Zeit der Änderungsoperation zu simulieren.

Mit der Anzahl der abzuarbeitenden Arbeits-Operationen kann festgelegt werden, wie viele dieser anderen Operationen ein Thread abarbeiten soll, bevor eine Modifikation zurückgeschrieben wird. Dazu wird zwischen dem Erhalten der Zustandsreferenz vom StateManager und dem Zurückschreiben der modifizierten Version eine Schleife welche diese Operationen beinhaltet aufgerufen.

Hierbei ist es bei der Durchführung von Messungen wichtig, dass der Thread tatsächliche Arbeit ausführt. Falls ein Thread nur mit der Thread.sleep() Methode angehalten werden würde, würde ein anderer Thread während dieser Zeit ausgeführt werden. Dies würde aber die Simulation unrealistisch gestalten, da Anwendungen simuliert werden sollen, bei denen die tatsächliche Modifikation eines Zustandes Zeit benötigt.

Listing 6.5: Zeit begrenzte Schleife

```
1 long startTime = System.currentTimeMillis();
2 while (System.currentTimeMillis() - startTime < timeLimit){}
```

Ein anderer Ansatz wäre wie in 6.5 die Threads eine bestimmte Zeit lang in einer Schleife zu halten, die einen leeren Körper besitzt. Wenn allerdings die Schleife mit der Differenz zwischen Schleifenstart und der aktuellen Systemzeit kontrolliert wird, kommt es zu einem Problem, wenn mehr Threads ausgeführt werden als Kerne im System vorhanden sind. Bei einer solchen Konstellation könnte ein Thread A durch das Betriebssystem unterbrochen werden und ein anderer ausgeführt werden. Wenn wieder zu Thread A zurück gewechselt wird, kann zwar unter Umständen die Schleife abgebrochen werden, da die Differenz zwischen Systemzeit und Startzeit groß genug ist. Jedoch muss dann beachtet werden, dass der Thread effektiv nicht so lange ausgeführt wurde und somit in einer realen Anwendung keine Arbeit verrichten konnte.

In der Beispielsdomäne wurde deshalb eine Möglichkeit implementiert gleichbleibende Operationen durchzuführen, die tatsächlich zur Auslastung des Prozessors führen und damit reale Aufgaben simulieren können.

Über der Anzahl der auszuführenden Operationen kann flexibel die Dauer der Modifikation simuliert werden. Die ausgeführten Operationen bestehen immer aus dem selben Code und beinhalten keine von außen zu steuernden Parameter. Dadurch wird immer die gleiche Arbeit verrichtet.

6.6.2 Ergebnisse

In der vorgestellten Beispielsdomäne wurden verschiedene Messungen durchgeführt, um die Performanz des zusammenführenden Ansatz zu untersuchen. Dabei wird die Performanz mit dem Ansatz manueller Synchronisation verglichen und untersucht welche Faktoren die Performanz des Zusammenführens beeinflussen.

6.6.2.1 Vergleich mit manueller Synchronisation

Im Vergleich mit manueller Synchronisation werden für jedes Verfahren mehrere Threads gestartet, die jeweils gleiche Aufgaben erledigen müssen bevor sie sich beenden. Gemessen wird die Zeit die vom langsamsten der Threads benötigt wird. Im Versuch beträgt die Arbeitsoperationendauer für 500 Arbeitsoperationen etwa 0.25 ms auf dem Rechner, wenn sie nicht unterbrochen wird. Die Änderungsoperationen werden so oft wie konfiguriert bei jeder Modifikation durchgeführt. Die Änderungen werden hierbei immer an der selben Stelle im Testobjekt vorgenommen. Dies führt dazu, dass nach jeder Änderungsoperation in der zusammenführenden Methode rekursiv die Objekte zusammengeführt werden müssen. Dies stellt ein Extremfall dar, da immer der gleiche Bereich modifiziert wird.

Dieser Versuch wurde mit verschiedener Anzahl von Threads unternommen, welche alle auf einen geteilten Zustand, der von dem Statemanager verwaltet wird, zugreifen und auf diesem Änderungen durchführen. Die Messungen wurden mit dem Zusammenführungs-Verfahren und zum Vergleich mit expliziter Synchronisation durchgeführt.

In 6.4 wurde der Versuch ohne zusätzliche Arbeitsaktion und mit 50 Arbeitsoperationen pro Modifikation durchgeführt. Hierbei fällt auf, dass die maximale Gesamtzeit um die Modifikationen durchzuführen, bei der Methode 2 deutlich geringer ausfällt, während die Zeiten von Methode 1 stark schwanken. Die Schwankungen sind wohl durch äußere Einflüsse wie das Scheduling des Betriebssystems bedingt, wie in der

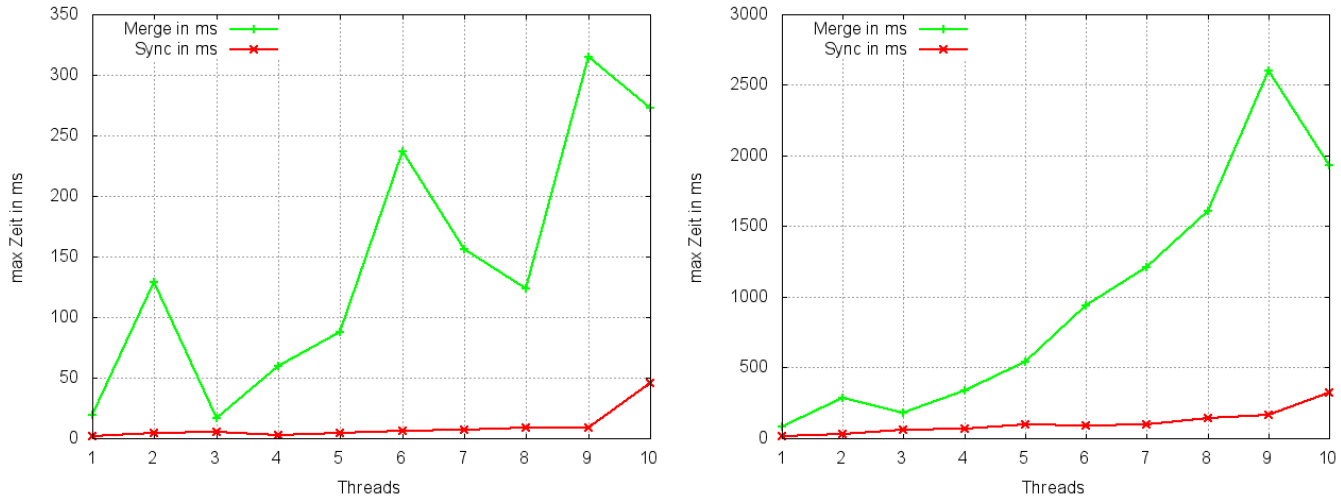


Abbildung 6.4: Links ohne Arbeitsoperationen, rechts mit 50. 1000 Modifikation pro Thread, Modifikationen in Tiefe 4 bei einer Gesamttiefe von 6

rechten Grafik zu sehen ist, stabilisieren sich die Zeiten etwas, wenn mehr Zeit für die Modifikationen benötigt wird.

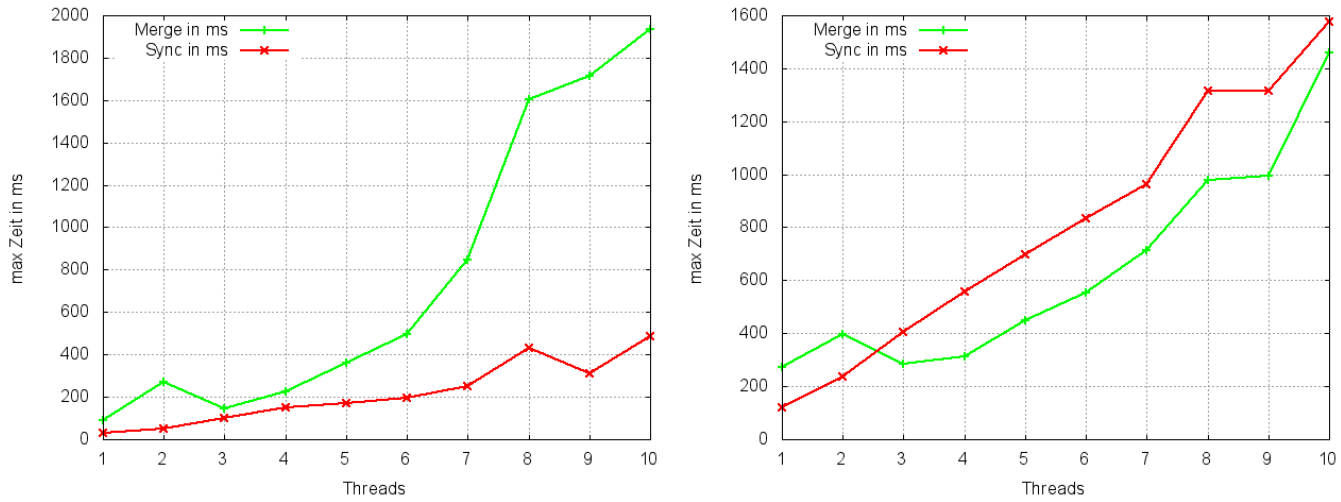


Abbildung 6.5: Links mit 100 Arbeitsoperationen, rechts mit 500. 1000 Modifikation pro Thread, Modifikationen in Tiefe 4 bei einer Tiefe von 6

Für 6.5 wurden die gleichen Messungen mit 100 und 500 Arbeitsoperationen durchgeführt. Hier ist interessant, dass der Versuch mit 100 Arbeitsoperationen länger braucht, als der Versuch mit 500 Arbeitsoperationen. Dies liegt an der Wahrscheinlichkeit mit der Objekte zusammengeführt werden, die beim Versuch mit 100 Arbeitsoperationen höher ist als beim Versuch mit 500. Dies zeigt, dass die Performance mit hoher Wahrscheinlichkeit von Konflikten einbricht. Dies korreliert auch mit dem Einbruch der Performance mit höherer Anzahl der Threads. Auf dem vier

Prozessorkerne enthaltende System, werden ab 5 Threads die Zusammenführungsoperationen öfter benötigt, da nun wenn das Setzen des neuen Zustandes blockiert, weitere Threads Änderungen ausführen.

Im Versuch mit 500 Arbeitsoperationen ist zu sehen, dass die Performance des zusammenführenden Verfahrens zum ersten Mal besser ist, als die der Methode mit expliziter Synchronisation. Das heißt ab einer Bearbeitungszeit von 0.25 ms pro Änderung lohnt es sich das Framework einzusetzen.

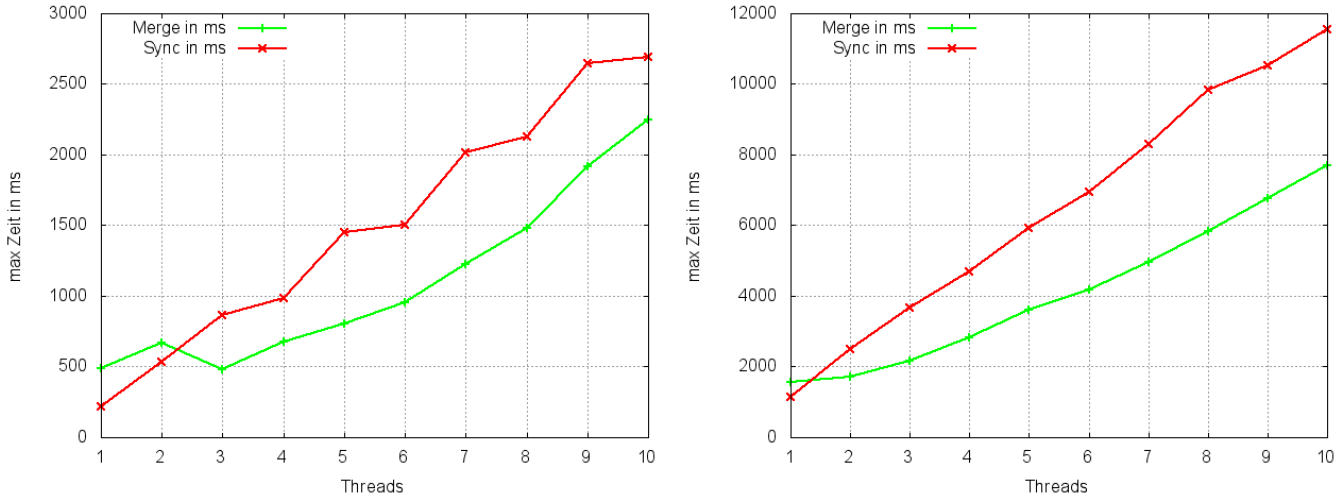


Abbildung 6.6: Links mit 1000 Arbeitsoperationen, rechts mit 5000. 1000 Modifikation pro Thread, Modifikationen in Tiefe 4 bei einer Tiefe von 6

Bei Versuchen mit mehr Arbeitsoperationen wird der Performanceunterschied deutlicher. Die Laufzeit steigt bei beiden Verfahren bei Erhöhung der Anzahl der Threads annähernd linear. Allerdings steigt sie bei der Zusammenführenden Variante weniger steil an.

6.6.2.2 Objektmerger interne Performanz

Da der Objektmerger auf die Reflection API aufbaut, die oft als sehr langsam bezeichnet wird, soll untersucht werden welche Teile des Objektmergers die meiste Laufzeit benötigen.

Hierbei werden Messungen mit dem YourKit Java Profiler durchgeführt, welcher es ermöglicht durch Sampling die benötigte Laufzeit pro Methode zu ermitteln.

Gemessen wurde ein Zusammenführen von PersistentTreeStructures der Tiefe 6, welche gegenüber einem gemeinsamen Vorgänger Modifikationen im gleichen Feld in der Tiefe 4 durchgeführt wurden. Dieses Zusammenführen wurde 5.000.000 mal durchgeführt, was zu einer Gesamtlaufzeit von ca. 15 Sekunden führte.

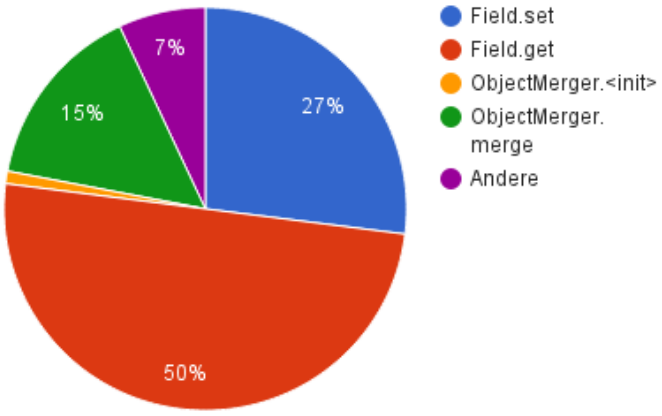


Abbildung 6.7: Aufteilung der Laufzeit innerhalb des Objectmergers

In 6.7 wird gezeigt, dass die meiste Zeit in der Methode Field.get() der Reflection API verbraucht wird. Diese wird dazu benutzt die einzelnen Felder der zusammenzuführenden Versionen und des gemeinsamen Vorgängers zu bekommen, und muss für jedes Feld der Objekte und deren Unterobjekte pro zusammenführen einmal aufgerufen werden. Dies lässt sich nicht vermeiden, da die Werte dazu benötigt werden um einen neuen Zustand mit den durchgeführten Änderungen zu erstellen.

Die am zweit aufwändigste Methode, ist mit Field.set() die Methode, die in dem Resultatsobjekt die neuen Werte setzt. Die aufwändigste Methode des ObjectMerger ist die ObjectMerger.merge Methode. In ihr werden die benötigten Informationen über die Felder einer Klasse ermittelt, und über die Felder iteriert um Änderungen festzustellen.

Die Methoden der Reflection API sind also zum Großteil für die Performanz des Zusammenführens verantwortlich. Allerdings ist die Nutzung nur auf Methoden beschränkt, welche durch den Ansatz direkt bedingt sind. Alle Informationen, die oft benötigt werden, werden in der ReflectionUtils Klasse zwischengespeichert. So müssen Felder und Informationen zu ihnen nur einmal pro Klasse ermittelt werden, was große Zeitersparnisse von mehreren Millisekunden pro Zusammenführungsoperation ermöglicht. Gerade bei Bäumen mit gleichem Typ der Knoten kann hier viel Zeit gespart werden.

6.6.2.3 ObjectMerger - Einfluss Änderungstiefe

Die Performance des ObjectMerger und somit auch die Performance des Frameworks hängt stark davon ab, wie aufwendig das Zusammenführen der Objekte ist. Dabei ist der Aufwand an zwei Parameter gekoppelt: Die Anzahl von Konflikten, das heißt Felder die bei beiden Zuständen gegenüber dem Vorgänger geändert sind, sowie der Tiefe dieser Konfliktobjekten im zusammenführenden Objekt.

Tiefe der Änderung	Zeit in ms	Aufruf Merge Funktion
1	5525	11111000
2	489	1112000
3	51	113000
4	7	14000
5	9	5000

Tabelle 6.3: Zusammenführen von zwei Objekten mit Tiefe 6, 1.000 Durchläufe

In 6.3 wird dargestellt wie sehr die Performanz von der Anzahl der benötigten Zusammenführungsoperationen abhängt. Hier wurden wie bei der vorherigen Messung zwei Instanzen der PersistentTreeStructure Klasse mit einer Tiefe von 6 zusammengeführt, die beide Änderungen an der gleichen Stelle hatten. Durch die Änderungen an gleicher Stelle, werden die betroffenen Felder vom Framework betrachtet und es wird versucht diese zusammenzuführen, indem ihre Kinderelemente betrachtet werden. Da alle Objekte unter dem geänderten Objekt komplett geändert sind, steigt der Aufwand, je niedriger der modifizierte Level ist. In der dritten Spalte ist die Anzahl der Aufrufe der Merge-Funktion für 1.000 Durchläufe des Zusammenführens angegeben.

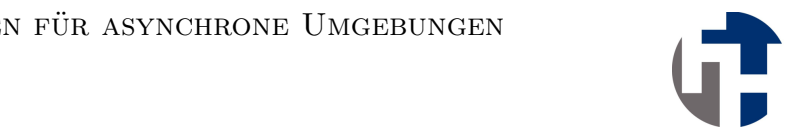
7 Audi Autonomous Driving Cup

Nachdem in den bisherigen Kapiteln auf die allgemeinen Eigenschaften von persistenten Datenstrukturen eingegangen wurde, sowie auf eine Implementierung eines Zustands verwaltenden Frameworks in Java, soll in diesem Kapitel die Benutzung von persistenten Datenstrukturen in einem konkretem Projekt untersucht werden. Hierfür wird die Implementierung eines geteilten Zustandes in asynchroner Umgebung im Rahmen des A_2O Projektes betrachtet, bei dem ähnlich wie beim entwickelten Framework ein gemeinsamer Zustand von mehreren Threads modifiziert wird.

7.1 Der Wettbewerb

Der Audi Autonomous Driving Cup ist ein von der Audi AG veranstalteter Wettbewerb, an dem Studententeams aus Deutschland teilnehmen können. Die teilnehmenden Teams mussten für eine vorgegebene, von Audi gestellte Hardwareplattform in Form eines Modellautos Software entwickeln, mit deren Hilfe das Auto vorgegebene Fahraufgaben autonom auf einer Modellstrecke absolvieren muss.

Das Modell ist ein akkubetriebenes Auto im Maßstab 1:8. Auf dem Auto sind verschiedene Sensoren verbaut, deren Werte von einem Arduino ausgelesen werden. Dieser Mikrocontroller schickt die Sensorwerte periodisch in pro Sensor verschiedenen Abständen zu einem auf dem Auto verbauten ODROID-X2 Board. Auf dem ARM basierenden ODROID Board läuft unter dem Betriebssystem Ubuntu die von Audi entwickelte Software ADTF (Automotive Data and Time-Triggered Framework). Mit ADTF werden die Sensorwerte entgegengenommen und verarbeitet. Befehle können über ADTF an das Arduino Board geschickt werden, welches die Werte umwandelt und über I^2C an die Aktuatoren schreibt.



Ziel des Wettbewerb ist es Komponenten für ADTF zu entwickeln, welche es ermöglichen die vorgegebenen Fahraufgaben möglichst zuverlässig und schnell zu absolvieren.

7.1.1 Hardware

Auf dem Auto sind verschiedene Sensoren verbaut, welche es den Teams ermöglichen sollen, die gestellten Aufgaben zu bewältigen.

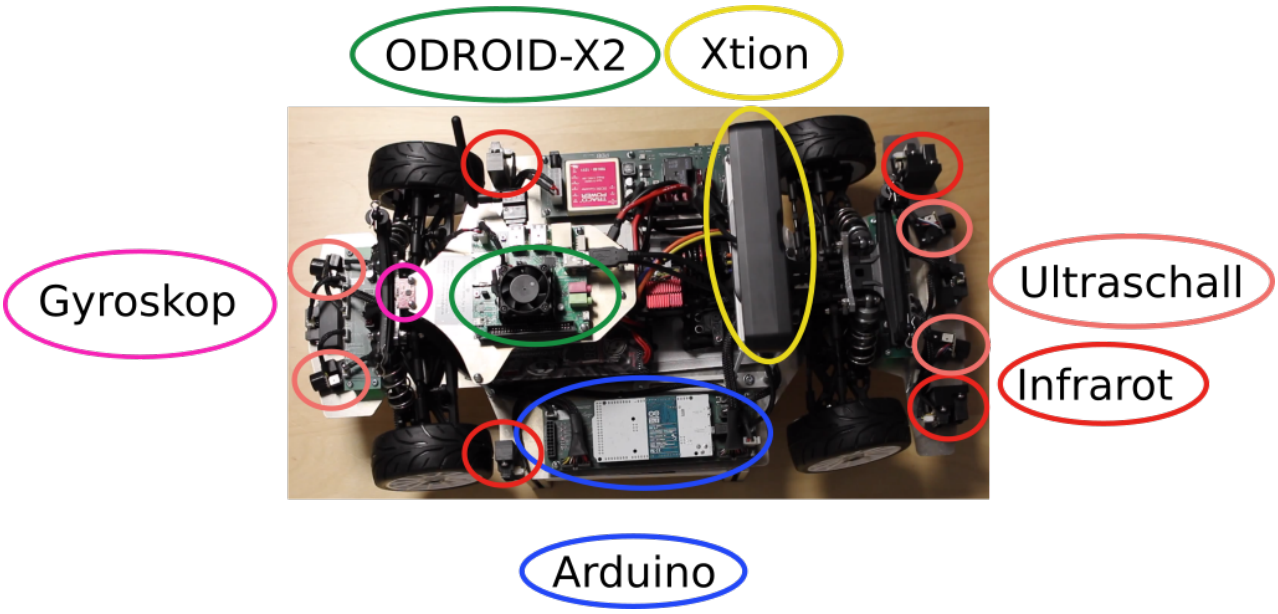


Abbildung 7.1: AADC Auto mit Sensoren

In Abbildung 7.1 ist das Automodell mit den verbauten Sensoren zu sehen. Das ODROID Board ist mit einem 1.7 GHz ARM Cortex-A9 Quad Core Prozessor ausgestattet. Mit Strom wird das Board entweder durch ein Netzteil versorgt, oder durch auf dem Auto verbaute Akkus. Die Sensordaten werden von einem auf dem Auto verbauten Arduino Mikrocontroller ausgelesen und über eine serielle Schnittstelle an den ODROID übertragen. Dort werden sie in ADTF durch einen Filter in ADTF Datenpakete gepackt, welche nach durchlaufen verschiedener Kalibrationsfilter an die Software des Teams geschickt werden. Die Software für das Entgegennehmen der Arduinopakete und die Arduino Software wurden von Audi geliefert und durften von den Teams nicht angepasst werden.

7.1.2 ADTF Framework

ADTF ist eine proprietäre Software Entwicklungs Plattform, die von Audi entwickelt und durch Elektrobit vertrieben wird. Hauptnutzer sind Automobil Herstel-

ler und ihre Zulieferer, die ADTF meist für die Entwicklung von Prototypen und Fahrassistenzsystemen benutzen. Die Software ermöglicht es durch eine graphische Benutzeroberfläche einzelne Komponenten miteinander zu verbinden und diese zu testen. Es können aufgezeichnete Sensordaten abgespielt werden, wodurch es möglich wird ohne Fahrzeug die Funktionalität der Softwarekomponenten zu entwickeln. Die in ADTF entwickelten Algorithmen müssen in der Serienfertigung nach Fertigstellung auf das konkrete Fahrzeug portiert werden. Hierbei müssen Abhängigkeiten zu ADTF entfernt werden, und der Code auf die konkrete Plattform portiert werden. Außerdem muss das Einlesen der Daten angepasst werden. Im Wettbewerb läuft ADTF direkt auf dem Fahrzeug, wodurch der Portierungsschritt entfällt und die Entwicklung vereinfacht wird.

ADTF baut auf das Konzept von Filtern auf, die Nachrichten entgegennehmen, diese verarbeiten und an andere Filter weiterschicken können. Diese Filter können graphisch durch den Anwender miteinander verbunden werden.

Filter besitzen kein oder mehrere Eingangspins und kein oder mehrere Ausgangspins. Pins stellen Möglichkeiten zur Verbindung mit anderen Filtern dar. Das Verbinden ist optional, auf nicht verbundenen Pins wird nichts empfangen. Die Pins werden in der Implementierung des Filters erzeugt und mit einem Datentyp versehen. Dadurch kann ADTF sicherstellen, dass nur kompatible Pins miteinander verbunden werden können. Die Datentypen sind Strukturen, welche vom Nutzer definiert werden müssen. Über Pins werden sogenannte MediaSamples verschickt und empfangen, welche diese Strukturen beinhalten. Sie werden über die Verwendung von ADTF Funktionen erstellt und gelesen.

Der Aufbau der MediaSamples ist durch sogenannte MediaDescription Dateien definiert. In ihnen wird durch eine xml Baumstruktur beschrieben wie die Struktur aufgebaut ist.

Listing 7.1 zeigt wie die Daten, die vom Arduino Filter geschickt werden, definiert sind:

Listing 7.1: Definition von MediaSample Daten

```
1 <struct alignment="1" name="tArduinoData" version="1">
2 <element alignment="1" arraysize="1" byteorder="LE" bytepos="0" name="ui8SOF" type="tUInt8" /
      >
3 <element alignment="1" arraysize="1" byteorder="LE" bytepos="1" name="ui8ID" type="tUInt8" />
4 <element alignment="1" arraysize="1" byteorder="LE" bytepos="2" name="ui32ArduinoTimestamp"
      type="tUInt32" />
5 <element alignment="1" arraysize="1" byteorder="LE" bytepos="6" name="ui8DataLength" type="
      tUInt8" />
6 <element alignment="1" arraysize="25" byteorder="LE" bytepos="7" name="ui8Data" type="tUInt8"
       />
7 </struct>
```

Auch die Typen wie tUInt8 werden in der Datei definiert. Durch das festlegen des Alignments und der Byteorder kann sichergestellt werden, dass die Daten auf allen Plattformen gleich strukturiert werden. In einem Filter können diese Strukturen nicht manuell angelegt werden, sie müssen über ADTF geschrieben werden, welches sicherstellt, dass die Byteorder und das Alignment eingehalten werden.

7.1.3 Fahraufgaben

Die Fahraufgaben werden vor dem Wettbewerb in Form einer xml-Datei, die eine Liste der Kommandos beinhaltet bereitgestellt. Es gibt sieben verschiedene Kommandos die berücksichtigt werden müssen. Die beim Wettbewerb umzusetzenden Kommandos sind in Tabelle 7.1 aufgelistet.

pull_out_left	Nach links ausparken, losfahren
pull_out_right	Nach rechts ausparken, losfahren
straight	geradeaus
left	links
right	rechts
parallel_parking	Seitlich parken
cross_parking	Quer parken

Tabelle 7.1: Kommandos im Wettbewerb

Die Richtungskommandos beziehen sich jeweils auf die Entscheidung an einer Kreuzung, in welche Richtung weitergefahren werden soll. Falls es nicht möglich ist, an der aktuellen Kreuzung das Kommando umzusetzen, muss es ignoriert werden. Wenn das Auto beispielsweise an einer Kreuzung steht, bei der es nicht erlaubt ist nach rechts einzubiegen (wegen eines Verbotsschildes, oder keiner Fahrbahn nach rechts), gilt das Kommando nicht für diese Kreuzung sondern für die nächste, bei der das Kommando eine gültige Option ist. Im Wettbewerb wird immer mit einem der Auspark-Kommandos begonnen. Die Kommandos werden vor einer Fahrt in Form einer xml Datei angegeben.

7.2 A_2O Architektur

Für die Umsetzung der Aufgaben wurde von unserem Team beschlossen eine Architektur zu entwickeln die wenig Abhängigkeiten zu ADTF besitzt. Außerdem soll auch das Auto so abstrahiert werden, dass eine Portierung auf ein anderes Fahrzeug möglich ist. Dabei wird eine Schichtenarchitektur eingesetzt, welche das Testen der einzelnen Komponenten ermöglicht und Portierungen einfacher macht. So kann

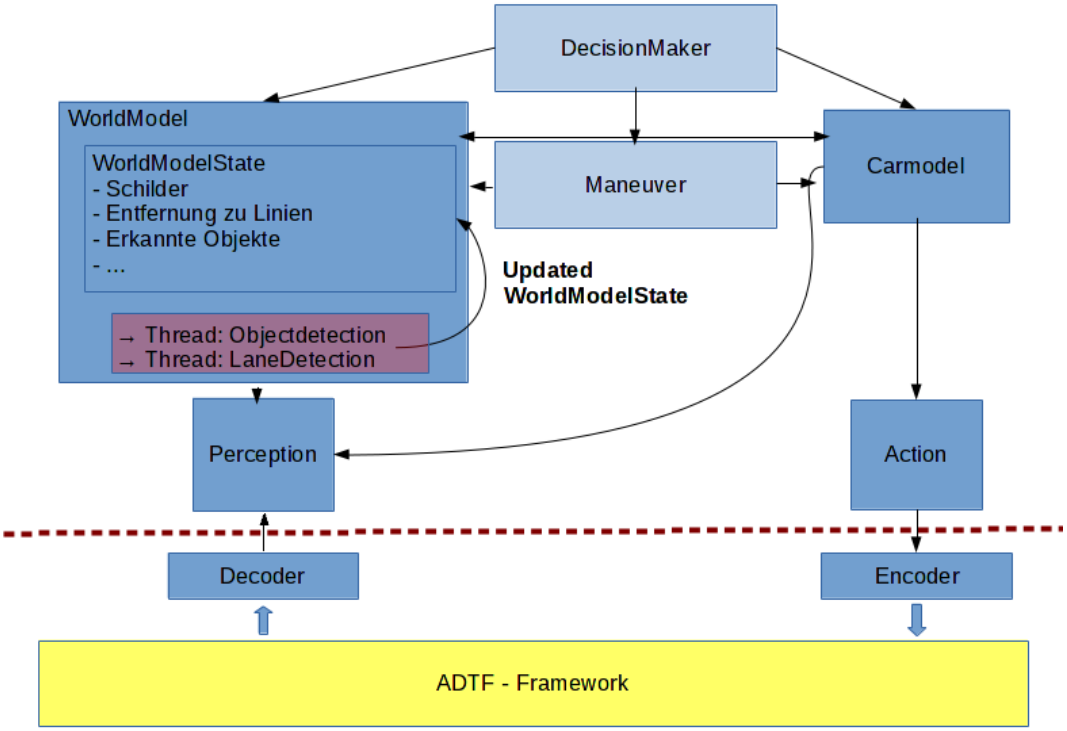


Abbildung 7.2: A_2O Architektur

die Verarbeitung der Sensordaten unabhängig von ADTF simuliert werden, entweder durch das Erstellen einer neuen Kommunikationsschicht, die Sensordaten durch andere Wege bereitstellt oder durch das Verwenden von aufgezeichneten oder simulierten Sensordaten. Als Sprache kommt C++ zum Einsatz, da die Software in ADTF integriert werden muss, was durch C++ am einfachsten durchgeführt werden kann. ADTF Filter sind unter Linux dynamisch gelinkte Shared Objects die beim Starten des Frameworks von Pfaden, die in den Einstellungen hinterlegt sind, geladen werden. Durch die Verwendung von C++ können die entwickelten Komponenten einfach in ADTF geladen werden, indem die gesamte entwickelte Software als Bibliothek kompiliert wird.

In Abbildung 7.2 werden die Hauptkomponenten der Software dargestellt.

Um die Abhängigkeit zu ADTF zu verringern und trotzdem auf alle Sensordaten zugreifen zu können, wurde ein einzelner ADTF Filter entwickelt, welcher alle verwertbaren Daten über Pins entgegennimmt und diese in ADTF unabhängige Datenstrukturen einliest. Dadurch wird der Vorteil der erzwungenen funktionalen Modularisierung nicht genutzt, die die Verwendung von einzelnen Filtern mit sich bringt. Allerdings ist die Modularisierung durch klare Trennung von Komponenten und dem Nutzen von Interfaces auch auf einer höheren Ebene der Architektur umsetzbar.

Die Architektur ist in verschiedene Schichten unterteilt, die voneinander durch die Verwendung von Interfaces getrennt sind. Die Kommunikationsschicht stellt die unterste Schicht dar, in ihr werden Sensordaten eingelesen und Aktor-Daten an die Hardware gesendet. Dabei werden von einem Thread die Pakete des ADTF-Frameworks entgegengenommen und durch den Decoder in ADTF-unabhängige Datenstrukturen konvertiert. Die decodierten Daten werden dann in die jeweiligen SensorObjekte des Perception Modules geschrieben. Hier endet die Verantwortlichkeit dieses Threads. Nach jeder Entgegennahme von Paketen im ADTF wird dieser Prozess durchlaufen.

Ein weiterer Thread ruft in einem festgelegten Zyklus die Update-Methode des CarModels auf. In ihr werden die Werte aus der Perception ausgelesen, die dort in generischer Form gehalten werden. Sie werden den passenden Sensor Objekten im CarModel zugewiesen und in ihnen abgelegt. Danach wird die Update-Methode des Worldmodels aufgerufen. In ihr werden nun die im CarModel abgelegten Sensorwerte verarbeitet. Es kann auch direkt auf das Perception Modul zugegriffen werden, falls Daten benötigt werden welche aus konzeptionellen Gründen nicht im CarModel gehalten werden. Im Wettbewerb sind dies beispielsweise die Kommandos des Jurymodules.

Asynchron zum update-Zyklus wird in einem weiteren Thread die Entscheidungsfindung angestoßen. Hierbei wird durch einen Decisionmaker die Entscheidung getroffen welches Maneuver aktuell gefahren werden soll. Er trifft dabei Entscheidungen aufgrund der aktuellen Situation in der sich das Auto befindet und den aktuellen Vorgaben der Jury. Die Maneuver berechnen aufgrund des WorldModelStates Parameter die im CarModel gesetzt werden. Sie versuchen hierbei bestimmte Ziele zu erreichen. So kann ein Maneuver versuchen das Auto langsam fahren zu lassen und dabei innerhalb der Fahrbahnmarkierung zu halten. Dafür muss es sicherstellen dass der korrekte Lenkwinkel gesetzt wird, welcher sich durch die Umgebung bei jedem Aufruf leicht ändert. Außerdem muss der Motor mit der gewünschten Geschwindigkeit angesteuert werden. Nach dem Ausführen des Maneuvers werden die nun in den Aktuatoren des CarModels hinterlegten Werte in Effektoren geschrieben. Diese Effektoren werden dann durch die Encoder Komponente, die von ADTF abhängig ist, ausgelesen um ADTF kompatible Nachrichten zu erzeugen. Die MedieSamples werden dann von ADTF an andere Filter weitergeleitet.

Für den Wettbewerb wurde ein zusätzlicher ADTF Filter benutzt, der die vom Hauptfilter verschickten Pakete jedes Pins untersucht und mit vorherigen Werten des Pins vergleicht. Wenn keine Änderung festgestellt werden kann, wird das Paket nicht weitergeleitet. Da nur Änderungen übertragen werden, kann der Verlust eines Paketes fatale Folgen haben. Um sporadisch auftretende Paketverluste bei der Über-

tragung zum Arduino auszugleichen, wird von dem weiterleitenden Filter außerdem in einer festgelegten Frequenz Pakete weitergeleitet, obwohl sie keine Änderungen beinhalten. Dies wurde durch Beschränkungen des Arduinos nötig, der nicht in beliebiger hoher Frequenz Pakete annehmen kann und bei zu hoher Sendefrequenz abstürzt.

7.3 Grundlagen Persistentes Model

Durch die Verwendung eines persistenten Models soll erprobt werden, ob die Verwendung von persistenten Datenstrukturen in einem asynchronen Umfeld Sinn macht. Außerdem soll untersucht werden ob die Verwendung eines persistenten Modells in einer praxisnahen Umgebung umsetzbar ist. In der A_2O Architektur soll das Weltmodell, welches die Informationen zur Umgebung des Fahrzeugs enthält, als persistentes Objekt erstellt werden. Durch die Eigenschaft der Persistenz soll es möglich sein das Weltmodell durch mehrere Threads gleichzeitig zu modifizieren. Das ermöglicht beim AADC, dass alle vier Kerne des ODROID ausgelastet werden können.

Aufgaben wie die Spur- und Objekterkennung, welche durch die Auflösung der Bilder und der Aufnahmefrequenz relativ viel Rechenzeit benötigen, können in eigene Threads ausgelagert werden, die unabhängig voneinander auf dem Weltmodell arbeiten. Dadurch wird der Hauptthread nicht blockiert und neue Sensordaten können entgegengenommen und verarbeitet werden, ohne von den anderen Aufgaben behindert zu werden. Das Weltmodell ist für die Verwendung von persistenten Datenstrukturen geeignet, da die verarbeitenden Threads recht lange Laufzeiten benötigten. Wie in vorherigen Kapiteln aufgezeigt, wird dadurch der zusätzliche Aufwand des Zusammenführens der Zustände weniger relevant.

Bei anderen Teilen der Architektur würde die Implementierung als persistente Datenstruktur kein Sinn machen. Im CarModel ist die Rate der Änderungen beispielsweise viel zu hoch, da alle neuen Sensorwerte in ihm abgelegt werden. Dies würde zu vielen erzeugten CarModel-Objekten führen, von dem jedoch in den meisten Fällen nur die letzte Version benötigt wird. Bei einer Umsetzung in der bei teilweisen Änderungen auf Teile der alten Versionen verwiesen wird, könnte zwar auch hier Speicher gespart werden, allerdings werden in fast jedem Updatezyklus ein Großteil der Sensorwerte verändert. Dadurch wäre der Anteil der wiederverwendbaren Daten sehr gering.

7.3.1 Erwartete Vorteile

Durch die Umsetzung des Weltmodells als persistente Datenstruktur soll es möglich sein performant mit Hilfe mehrerer Threads das Weltmodell zu verändern. Außerdem soll die Umsetzung gegenüber einer Lösung mit traditionellen Synchronisierungsmechanismen wie Mutexe und Locks weniger fehleranfällig für Probleme wie Dead-Locks und Dataraces sein. Durch die Wiederverwendung von Objekten und der Vermeidung von tiefen Kopien soll die Performance akzeptabel gegenüber herkömmlichen Synchronisations-Methoden bleiben.

7.3.2 Erwartete Nachteile

Aufgrund von Spracheigenschaften von C++ ist es schwierig ein Framework zum Verwalten und Verwenden von WorldStates oder States im Allgemeinen zu entwickeln. Im Vergleich zu Java fehlt hier vor allem eine Technik, die Funktionen vergleichbar mit dem Reflection Framework bereitstellt. Deswegen wurde das in Kapitel 6 vorgestellte Framework in Java implementiert, womit die Praktikabilität eines generischen Frameworks erprobt wurde. Da es keine Möglichkeit gibt die Member Felder eines Objektes zur Laufzeit zu ermitteln, können nicht ohne Wissen über die interne Gestaltung der Member Felder Objekte zusammengeführt werden. Dadurch muss explizit im Code auf die konkreten Memberfelder zugegriffen werden, sonst gibt es keine Möglichkeit auf sie zuzugreifen.

Außerdem muss bei der Verwendung von Objekten darauf geachtet werden, dass diese entweder selbst unveränderbar implementiert sind, oder dass tiefe Kopien erstellt werden. Da im AADC Projekt viele Typen verschiedener Open Source Projekte verwendet werden, die teilweise eigene Speicherverwaltung beinhalten, ist dies nicht immer offensichtlich. Die verwendeten Eigen und OpenCV Bibliotheken stellen zum Beispiel Typen bereit, bei der Kopien auf die selben Speicherbereiche zeigen und tiefe Kopien explizit durchgeführt werden müssen. Dies steigert in vielen Anwendungsfällen die Performanz, kann aber zu Problemen führen wenn dieses Verhalten nicht beachtet wird. Dass die Member nicht dynamisch ermittelt werden, spart zwar Laufzeit, jedoch muss bei jeder Änderung am Code des Zustandes darauf geachtet werden, dass die Persistenzeigenschaft erfüllt bleibt.

7.3.3 Architektur

Ähnlich wie im entwickelten Framework wird auch in der C++ Umsetzung eine Verwaltungsklasse verwendet, welche den momentanen sowie die bisherigen Zustände verwaltet. Diese Aufgabe übernimmt die WorldModel Klasse, in der Informationen

zur Umgebung des Autos gehalten und verarbeitet werden. In ihr wird ein Pointer auf ein unveränderbares WorldState Objekt gehalten. Außerdem werden in einem WorldStateHolder die bisherigen Zustände gespeichert.

Verschiedene Threads, die asynchrone Aufgaben übernehmen, werden durch das WorldModel gestartet. Sie können über einen Pointer auf das WorldModel zugreifen. Die folgenden Aufgaben werden in eigenen Threads abgearbeitet:

- Straßenerkennung
  Hier wird das 2D-Bild der Xtion Kamera verarbeitet. Aufgrund erkannter Linien werden Streckensegmente angelegt, welche in eine virtuelle Karte mit eingearbeitet werden.
- Objekterkennung
  Bei der Objekterkennung wird das 3D-Bild der Xtion Kamera, sowie die Werte der Ultraschall und Infrarot Sensoren verwendet. In den Daten werden physikalische Objekte im Sichtfeld der Sensoren gesucht und ihre Ausmaße vermessen. Die erkannten Objekte werden im WorldState angelegt.
- Schildererkennung
  Die Schildererkennung benutzt das 2D-Bild der Kamera, um es auf vorgegebene Schilder zu untersuchen. Sie werden erkannt, ihre globale Position bestimmt und im WorldState abgelegt.

Durch die Abarbeitung dieser Aufgaben in eigenen Threads ist es möglich wichtige, schnell ablaufende Prozesse im Hauptthread durchzuführen. Dies sind beispielsweise die Positionsbestimmung des Autos im Raum, sowie die Verarbeitung von Kommandos und die Entscheidungsfindung auf Grund der aufgebauten Informationen über die Umgebung. Wenn die aufgelisteten Aufgaben alle im Hauptthread laufen würden, wäre es nicht möglich in hoher Frequenz Aktionen an die Aktuatoren zu senden. Außerdem käme es Aufgrund großer Schwankungen der Laufzeit in den bildverarbeitenden Algorithmen zu Schwankungen in der Reaktionszeit. So könnte ein Bereich der Bildverarbeitung das gesamte System verlangsamen. Auch Sensorsdaten könnten verloren gehen, da ADTF die MediaSamples nicht unbegrenzt puffert.

Das WorldModel stellt die Methode setNewWorldState bereit, mit der die Threads ihre modifizierten Zustände wieder in das WorldModel überführen können, damit die anderen Threads im nächsten Zyklus auf die neuen Informationen zugreifen können. In der Methode werden die beiden Zustände zusammengeführt. Dies geschieht wie im entwickelten Framework über den Vergleich des momentanen Zustandes, des neuen Zustandes und ihrem gemeinsamen Vorfahren. Die Versionierung wurde, wie in der Java Version, über verlinkte VersionInfo Objekte realisiert, die in den Zuständen enthalten sind. Die WorldStateHolder Klasse verwaltet die bisherigen Zustände und

ermöglicht es eine maximale Anzahl von gespeicherten Zuständen festzulegen. Dies wird genutzt, um während der Laufzeit des Programmes nicht immer mehr Speicher zu benötigen.

7.4 Implementierung

Bei der Implementierung des WorldStates werden einige Eigenschaften von C++ genutzt, welche die Umsetzung von unveränderbaren Objekten vereinfachen.

Der WorldState stellt für die beinhalteten Daten Methoden bereit, mit denen diese verändert werden können. Hierbei wird ein neues WorldState Objekt erzeugt, welches dem Aufrufer zurückgegeben wird. Hierbei ist es wichtig, den bisherigen WorldState nicht zu modifizieren. In C++ kann dies einfach realisiert werden, in dem alle Methoden der Klasse mit dem const Qualifier markiert werden. Dadurch können die Veränderungsmethoden nur von mit const markierten Instanzen verwendet werden. Außerdem verhindert der Compiler, dass mit const markierte Methoden das Objekt modifizieren, wodurch schon beim Entwickeln Fehler vermieden werden. Somit kann sichergestellt werden, dass ein Objekt unveränderbar implementiert ist. Die Empfehlung auch die Member mit const zu markieren, ist nicht immer umsetzbar. Bei der Implementierung des PersistentenVectors sind beispielsweise die Änderungsoperationen nicht mit const markiert. Obwohl diese das Objekt nicht verändern, sondern ein neues Objekt mit den gewünschten Änderungen zurückliefern, können dadurch keine Änderungsoperationen auf mit const markierten Objekten durchgeführt werden. Diese Einschränkung beeinträchtigt die Robustheit des Codes und führt zu Unklarheiten bei der Analyse des Codes. Bei der Verwendung von vielen Bibliotheken kommt es oft zu solchen Problemen, da die vollständige Markierung des Codes mit const oft nicht von den Entwicklern durchgeführt wird. Dadurch ist es bei jedem Typ nötig die Implementierung zu betrachten, um sicherzustellen ob die Objekte unveränderbar implementiert sind.

7.4.1 Transiente Builderklasse

Intern verwendet die WorldState Klasse eine transiente Builder Klasse. Wie in Grundlagen dargestellt wird, ermöglicht es eine transiente Klasse Änderungen durchzuführen, ohne dass für jeden Einzelschritt aufwändige Operationen durchgeführt werden müssen.

Die transiente WorldStateBuilder Klasse wurde hierbei als interne Friend Klasse des WorldStates implementiert. Das bedeutet, dass kein Nutzer des WorldStates auf sie Zugriff hat, in der Builder Klasse aber auf private Member des WorldStates

zugegriffen werden kann. Eine weitere Besonderheit ist, dass die transiente Klasse vom WorldState erbt. Dadurch hat sie die gleichen Member, was ihre Benutzung als Builder vereinfacht.

Im Konstruktor des Builders werden die Memberfelder mit den Werten der jeweiligen Felder des zu modifizierenden Zustandes initialisiert. Auf dem transienten Objekt können dann mehrere Update-Operationen ausgeführt werden, ohne dass neue Objekte des WorldStates erzeugt werden müssen. Durch das Aufrufen der persist-Methode wird ein neues persistentes Zustandsobjekt mit den momentan im Builder vorhandenen Daten erzeugt. Dieses Vorgehen verbessert bei aufwändigen Update-Operationen die Performance.

Außerdem wird die Lesbarkeit des Codes verbessert, da in den Änderungsmethoden des WorldStates der direkte Konstruktor Aufruf vermieden werden kann. Dies wird vor allem relevant wenn eine Zustandsklasse viele Memberfelder enthält.

Listing 7.2: Verwendung des transienten Builders

```
1
2  // Benutzung des transienten Builders
3  WorldState::ConstPtr WorldState::addSignResult(const SignResult toAdd) const
4  {
5      WorldStateBuilder builder(this);
6      builder.addDetectedRoadSign(toAdd);
7
8      return builder.persist();
9  }
10
11 // Ohne Benutzung des transienten Builders
12 WorldState::ConstPtr WorldState::addSignResult(const SignResult toAdd) const
13 {
14   return boost::make_shared<const WorldState>(_this,
15             _roadSigns.add(toAdd),
16             _currentSegment,
17             _detector,
18             // ... alle weiteren Member
19             );
20 }
```

In Listing 7.2 ist als Beispiel eine Update-Methode die den Builder verwendet, und eine Update-Methode die ohne Builder implementiert wurde dargestellt. Der Aufruf des Konstruktors bei der Variante ohne Builder müsste für jede Update-Operation wiederholt werden. Außerdem müssten bei jeder Einführung eines neuen Members alle Update-Methoden angepasst werden. Durch den Builder muss nur der Konstruktor des Builders, sowie die persist Methode angepasst werden.

7.4.2 WorldState

Bei der Implementierung der Zustandsklasse muss außerdem darauf geachtet werden, dass C++ Parameter von Funktionen als Wert entgegennimmt. Das heißt, dass bei jedem Funktionsaufruf Kopien der übergebenen Objekte erstellt werden. Wenn ein Zustand beispielsweise einen Vector von Objekten halten soll, würde jede Änderung am Zustand durch den Konstruktoraufruf eine neue Kopie des gesamten Vectors mit seinen Elementen verursachen. Dies kann bei großen Objekten zu Probleme mit dem benötigtem Speicher und auch zu Laufzeit Problemen führen. Beheben lässt sich dieses oft unerwünschte Verhalten durch die Verwendung von Pointern oder Smartpointern. Da Pointer innerhalb einer Plattform immer die gleiche Größe besitzen und Smartpointer nur wenig Speicher benötigen, sollten sie für jeden Teil des Zustandes genutzt werden.

Da im A_2O Projekt Smartpointer verwendet werden, bietet sich die Verwendung auch in der Implementierung des persistenten Zustands an. Alle Felder des Zustandes müssen hinter einem Smartpointer gekapselt werden, oder als Wert in einem Container wie dem PersistentVector gespeichert werden. Nur so kann vermieden werden, dass immer mehr Speicher bei Zustandsänderungen benutzt wird.Mit der Nutzung von Smartpointern wird, wenn ein Zustand nicht mehr in der Liste der bisherigen Zustände gehalten wird und kein anderer Zustand auf die Objekte des Smartpointers zeigt, der komplette Speicher des nicht mehr benötigten Zustandes automatisch freigegeben. Ohne die Verwendung müssten im Destruktors des Zustandes alle beinhalteten Objekte explizit freigegeben werden. Da aber nicht klar wäre, ob andere Zustände noch die Objekte verwenden, könnte Speicher nie freigegeben werden.

Das Zusammenführen muss wegen des Fehlen einer Technik zur Ermittlung der Felder eines Objektes zur Laufzeit konkret für die Zustandsklasse implementiert werden. Hierbei werden für alle Felder einzeln die benötigten Werte ermittelt. Mit den Werten des Feldes des neu einzupflegenden Zustands, des bisherigen Zustands und ihrem gemeinsamen Vorfahren kann dann ermittelt werden, ob und in welchem Zustand Änderungen am Feld ausgeführt wurden. Bei der Verwendung von Smartpointer auf konstante Objekte müssen nur die Werte der Pointer verglichen werden. Wenn ein Zustand ein verändertes Objekt beinhaltet, ist auch der Wert des Pointers geändert. Durch die Nutzung einer Template Methode kann das Ermitteln des für den neuen Zustand zu nutzenden Objektes für Smartpointer aller Typen generisch implementiert werden. Die Entscheidungen werden hierbei, genau wie im Java Framework, aufgrund von Tabelle 6.1 getroffen.

Bei Typen welche als Werte in den Zustandsklassen gehalten werden sollen, müssen die Objekte selbst verglichen werden. Da in C++ nicht davon ausgegangen werden

kann, dass Objekte durch einen == Operator oder eine equals Methode verglichen werden können, kann keine Template Implementierung verwendet werden.

In der Methode die das Zusammenführen übernimmt, wird ein Builder Objekt erstellt, das auf den momentanen Zustand basiert. Dann wird für jedes Feld des Zustandes der Wert für den neuen Zustand ermittelt, und dieser im Builder gesetzt. Nachdem aus dem Builder ein neues persistentes Objekt erzeugt wurde, wird das Objekt als aktueller persistenter Zustand im WorldModel gesetzt.

Die rekursive Zusammenführung wie in Java ist aus Gründen die im Kapitel Evaluation beschrieben werden, nicht möglich.

Dadurch bleibt nur die Möglichkeit der expliziten Konfliktbehebung. Dafür muss im Aufrufenden Code durch den Entwickler festgelegt werden wie der Konflikt behandelt werden soll. In vielen Fällen ist es nur möglich aufgrund von Richtlinien wie in Tabelle 6.1 zu entscheiden, da nicht auf interne Felder der Objekte zugegriffen werden kann.

7.4.2.1 Informationen im WorldState

In der WorldState Klasse sollen im A_2O Projekt Informationen über die Umgebung abgelegt werden. Aufgrund der Informationen im WorldState sollen durch die Entscheidungsschicht Aktionen abgeleitet werden. Bei den abzulegenden Informationen muss darauf geachtet werden, dass sie sinnvoll von der Entscheidungschicht benutzt werden können. Dabei muss ein Zwischenweg zwischen dem Halten aller verfügbarer Informationen und dem Halten nur der nötigsten Informationen gefunden werden. Dies ist durch das Halten vieler alten Zustände bedingt. So wäre es nicht umsetzbar, in jedem Zustand sowohl das letzte verarbeitete Kamerabild und die daraus berechneten Strecke zu halten, da dies zu einem zu hohen Speicherverbrauch führen würde. Außerdem sollte die Entscheidungsschicht nicht mit den unverarbeiteten Informationen arbeiten. Wenn dies vermieden wird, können die unteren Schichten einfacher ausgetauscht oder anders implementiert werden.

Im WorldState für das A_2O Projekt werden folgende Informationen gehalten:

- Erkannte Objekte
  Position und Struktur der physikalischen Objekte auf der Strecke.
- Aktuell aufgebaute Karte
  Struktur in der eine Karte aufgrund der erkannten Streckenteilen gehalten wird.

- Erkannte Schilder
  Typ und Position und Häufigkeit der Erkennung der erkannten Verkehrszeichen. Dabei werden die Ergebnisse aller bisher verarbeiteten Bilder verwendet.

7.5 Evaluation

Im Folgenden sollen die Erkenntnisse dargestellt werden, die während der Implementierung für das Projekt A_2O gewonnen wurden. Die praktische Durchführung des Versuches, Teile als persistente Zustände zu implementieren, zeigte viele Probleme des Konzeptes auf. Vor allem die Verwendung von C++ als Programmiersprache erschwert aufgrund fehlender Sprachunterstützung die Umsetzung.

7.5.1 Persistente Datenstrukturen in C++

Persistente Komponenten in C++ zu schreiben, wird zunächst durch das Fehlen vorhandener Implementierungen persistenter Basisdatenstrukturen erschwert. So wäre es in vielen Fällen nützlich gewesen, persistente Maps zu benutzten. Leider gibt es keine öffentliche Implementierungen persistenter Maps in C++, die nicht große Performance Nachteile gegenüber der Implementierung in der C++ Standardbibliothek besitzen. Außerdem kann durch das Fehlen von Typinformationen zur Laufzeit kein wiederverwendbares Framework entwickelt werden, dass flexibel genug für die praktische Nutzung ist.

Es ist jedoch durchaus möglich für konkrete Problemstellungen persistente Zustandsklassen zu implementieren und diese zu verwenden. Hierbei muss darauf geachtet werden, dass die verwendeten Klassen unveränderbar implementiert sind. Bei der Zusammenführung muss beachtet werden, dass Objekte die private Felder enthalten nur mit Anpassungen rekursiv zusammengeführt werden können.

Aufgrund der Komplexität mancher Objekte von externen Bibliotheken ist es oft sehr schwer eine Aussage über diese Unveränderbarkeitseigenschaft zu treffen. Außerdem ist es oft schwierig herauszufinden mit welcher Methode eine tiefe Kopie eines Objektes erzeugt werden kann. Die Verwendung der Überladung von Operatoren erzwingt das Betrachten des Codes bzw. der Dokumentation der Bibliotheken, um nachvollziehen zu können, welche Teile wie kopiert werden müssen.

Die Behandlung von Konfliktfällen beim Zusammenführen stellt ein weiteres Problem dar. In der Java Implementierung wurde dies durch die Verwendung von Annotationen vereinfacht. Dort konnte für jedes Feld entweder explizit angegeben werden welches Feld benutzt werden soll, oder eine Methode für die Behebung des Konfliktes

festgelegt werden. In C++ wäre es durch das Fehlen von Annotationen nur möglich für jeden Typ eine Methode festzulegen. Dazu müsste jede als Feld im Zustand verwendeten Klasse von einer gemeinsamen Oberklasse erben, welche eine abstrakte Methode zur Konfliktbehebung bereitstellt. Da aber nicht bei jeder Verwendung einer Klasse die Konfliktbehebung gleich ablaufen soll, ist dieser Mehraufwand aufgrund fehlender Flexibilität nicht praktikabel.

Die einfachste Variante unterschiedliche Methoden bereitzustellen, ist ihre Implementierung in der entsprechenden Klasse. Dabei wird in jedem Objekt ein Wert abgelegt, der die jeweils durchzuführende Konfliktbehebung identifiziert. Der Merge Code kann dann für alle Felder eines Types die gleiche Methode aufrufen, welche dann die korrekte Konfliktbehebung aufruft. Eine andere Variante wäre die Verwendung von Klassen die von der Basisklasse erben. Hierbei würde für jede Art der Konfliktbehebung eine neue Klasse erstellt, welche eine eigene Konfliktbehebungsmethode überschreibt. So wäre schon über den Typ eines Feldes erkennbar, wie es sich im Konfliktfall verhält.

Außerdem ist es nicht möglich wie in Java bei Konfliktfällen die Zusammenführung rekursiv durchzuführen. In C++ ist es nicht möglich auf die privaten Felder eines Objektes zuzugreifen. Die Verwendung von friend Klassen und Methoden ermöglicht zwar genau dies, jedoch müssen beide betroffenen Klassen die andere Klasse kennen. Das macht es unmöglich auf Felder von externen Klassen zuzugreifen, beispielsweise von Klassen aus verwendeten Bibliotheken, die nicht verändert werden können. Deswegen kann nicht immer, wie in der Java Implementierung, die Zusammenführung rekursiv ausgeführt werden. Um das Zusammenführen rekursiv durchführen zu können, muss für jeden Typ der Unterebene eine explizite Zusammenführungsfunktion geschrieben werden. Für externe Typen könnte dies durch die Verwendung von Klassen, die von den externen erben und diese durch die Zusammenführungsfunktion erweitern erreicht werden. Allerdings kommt es auch dann zu Problemen, wenn Objekte eines Types als Feld mit Typ einer Superklasse gespeichert werden. Es müsste für jeden Wert ermittelt werden, welcher konkrete Typ er besitzt, damit die korrekte Zusammenführungsfunktion verwendet werden kann. Aufgrund dieser Probleme ist eine robuste, verwendbare Implementierung der rekursiven Zusammenführung in C++ nicht umsetzbar.

Es ist also nur in sehr begrenzten Fällen sinnvoll, in C++ implementierte persistente Zustandsklassen in asynchronen Umgebungen zu benutzten. Sobald die Zustände viele verschiedene Typen beinhalten, wird die Implementierung der Zusammenführungsfunktion sehr komplex. Außerdem kann oft nur in der obersten Ebene des Zustandes zusammengeführt werden. Dies führt bei Konflikten zu großem Spei-

cherverbrauch, da durch jede Änderung an Teilen eines Unterobjektes das gesamte Unterobjekt kopiert werden muss.

In Projekten in denen es durch die Zuständigkeiten der Threads zu keinen Konflikten in den erzeugten Zuständen kommt, kann durch das Wegfallen der Konfliktbehandlung die Implementierung vereinfacht werden. In solchen Fällen muss jedoch genau betrachtet werden, ob die Vorteile der Umsetzung mit persistenten Zuständen die Nachteile der recht komplexen Implementierung überwiegen.

7.5.2 A_2O spezifische Probleme

Voraussetzung einer Implementierung von voll persistenten Zuständen ist, dass die enthaltenden Objekte als unveränderbare Objekte implementiert sind. Wenn diese Voraussetzung nicht erfüllt wird, kann durch die Modifikation eines Objektes das in einem Zustand referenziert wird, Zustände manipuliert werden. Die Zustände sind somit nicht persistent.

Im A_2O Projekt wurden die grundlegenden Datenstrukturen nicht als unveränderbare Objekte implementiert. Die im Zustand abgelegten Objekte sind mit Ausnahme der Karte recht einfach gestaltet. Die Objekte und Schilder enthalten nur einige Daten bezüglich der Position und des Typs. Eine Anpassung der Klassen, bei der diese als unveränderbare Klassen implementiert werden ist einfach durchführbar. Da von beiden eine Liste mit den bisher erkannten Objekten geführt werden muss, kann die Implementierung des PersistentVector benutzt werden.

Zu Problemen kommt es bei der Anpassung der Karteninformationen. Die Karte wird aus Segment Objekten aufgebaut. Diese beinhalten Informationen wie die Streckenbreite, die Länge des Segmentes, sowie die Krümmung. Aufgrund der Vorgaben des Wettbewerbs wurden die Segmente so modelliert, dass sich die Strecken des Wettbewerbs gut abbilden lassen. So gibt es Kurvensegmente, Kreuzungssegmente und Geradensegmente. Andere Elemente wie Kreisverkehre werden nicht berücksichtigt. Die Segmente werden durch sogennante SegmentLinks verbunden. Dabei kann ein Segment, je nach Typ eine verschiedene Anzahl von SegmentLinks beinhalten. Die SegmentLinks enthalten je einen Pointer auf das vorherige Segment als auch einen Pointer auf das angeschlossene Element. Außerdem enthalten die Segmente die für die Entscheidungsfindung relevanten Umgebungsinformationen, wie die Schilder die sich auf eine Kreuzung beziehen. An Segmente können weitere Segmente angebaut werden, in dem an einen der Segmentpunkte auf ein neues Segment verwiesen wird.

Da ein Segment SegmentLinks beinhaltet, welche wiederum einen Verweis auf das Segment besitzen, ist es nicht möglich die beiden Klassen unveränderbar zu imple-

mentieren. Dies ist durch die Notwendigkeit bedingt, bereits im Konstruktor des jeweiligen Elementes die Speicheradresse des anderen Segmentes zu kennen.

Deswegen muss es durch Modifikationen ermöglicht werden, den Baum der von den Segmenten aufgespannt wird, zu modifizieren damit ein neuer Baum mit der gewünschten Änderung zurückgegeben werden kann. Die einfachste Möglichkeit dies zu bewerkstelligen ist die Entfernung eines der Pointer des Segmentlinkes. Da der nach vorne zeigende Pointer für beinahe alle Berechnungen bezüglich der Karte benutzt wird, kann er nicht entfernt werden. Ohne ihn wäre es nicht möglich von einer globalen Position des Autos die Position auf der virtuellen Karte zu bestimmen. Der auf das vorherige Segment zeigende Pointer kann dagegen relativ einfach entfernt werden. Dadurch ist es möglich Segmente an unveränderbare Segmente anzuhängen, und einen neuen Segment-Baum zu erhalten.

Auf der aus den Segmenten bestehenden Karte werden die abzufahrenden Wegpunkte berechnet. Die Wegpunkte werden dafür aufgrund der Positionen der Segmentlinks berechnet, welche angefahren werden sollen. Dies ist nötig, um rechtzeitiges und korrektes einlenken bei Abbiegemaneuver zu ermöglichen, da die Information wo die Strecke in neue Segmente abzweigt, in den SegmentLinks und Segmenten enthalten ist. Auch dass sich die Befehle des Jurymodul immer nur auf das nächste physikalische Streckensegment bezieht indem der Befehl gültig ist, muss in die Planung aufgenommen werden.

Zu Problemen mit der Umsetzung der Unveränderbaren Segmente kommt es bei der Berechnung ihrer Position. Bei den Berechnungen der Spurerkennung werden je nach dem Abstand des Autos vom nächsten und der Position auf dem momentanen Segment durch Messungenauigkeiten andere Abstandswerte zum nächsten Segment ermittelt. Hierbei sind die Messungen in einem bestimmten Bereich präziser als in anderen. Da es für die Aufgabenstellung extrem wichtig ist die korrekte Position der Straße zu erkennen, kann nicht einfach die erste Erkennung eines Segmentes übernommen werden. Dies könnte zu Fehlern führen, falls ein Segment falsch erkannt wurde, oder die Position des Segmentes zu ungenau ist. Deshalb ist es nötig sowohl für den Typ der Segmente als auch für die Position und Länge der Segmente die Resultate möglichst vieler relevanter Messungen mit einfließen zu lassen. Hierbei muss der Bereich, indem die Berechnungen am zuverlässigsten sind, einen größeren Einfluss verglichen zu anderen Bereichen haben. Das Problem besteht hierbei bei Update-Operationen der Segmente, bei denen die Position geändert werden soll. Hierbei muss durch die Unveränderbarkeitseigenschaft der Segmente, die Position des vorherigen Elementes bereits bekannt sein. Dadurch, dass ein Segment allerdings auch die von ihm verwendeten SegmentLinks und somit auch sein nachfolgende Elemente kennen muss, kommt es zu einem Konflikt. Das neue Segment mit

neuer Position kann nicht erstellt werden, da keine nachfolgenden Segmente mit korrekter Position erzeugt werden können. Diese würden nämlich sowohl die ursprüngliche Verschiebung als auch die Position aller vorherigen verschobenen Segmente benötigten, um ihre eigene Position zu berechnen. Das Problem kann durch die temporäre Berechnung aller Segmentpositionen, ausgehend von dem ersten Segment gelöst werden. Hierbei wird im Segmentbaum in jeder Ebene für die enthaltenen SegmentLinks ein temporäres Objekt mit verschobener Position erzeugt. Wenn ein Segment auf keine weiteren Segmente verweist, ist es als final anzusehen und kann von dem temporären Segment der höheren Ebene genutzt werden, um ein finales unveränderbares Segment zu erzeugen. Dieses Verfahren muss rekursiv ausgeführt werden, bis das Startsegment mit finalen Verweisen auf die benachbarten Segmente erzeugt werden kann. Aufgrund der Komplexität der Änderungen, die benötigt würden damit dieses Verhalten umgesetzt werden kann, wurde keine Implementierung durchgeführt.

Bei anderen im Zustand gekapselten Daten, lässt sich die Unveränderbarkeit einfacher implementieren. So können die wenigen Änderungsoperationen der Schilder leicht angepasst werden. Alle bisher erkannten Schilder werden im Zustand gespeichert. Sie beinhalten eine Position im Raum, den Typ des Schildes, sowie einen Zähler der zählt wie oft das Schild erkannt wurde. Dies ist nötig um Fehlerkennungen zu vermeiden, bei denen bei der Verarbeitung eines einzelnen Bildes ein Schild fälschlicherweise erkannt wurde. Pointer auf die Schilder werden in einem PersistentenVector abgelegt. Hier wird der Persistenzvorteil der Implementierungen ausgenutzt. Durch die Persistenz können beliebig viele Threads über einen Zustand auf die Schilder zugreifen. Dabei kann die Version des Zustandes schon nicht mehr die aktuellste sein und trotzdem bleibt der Zugriff auf den Vector abgesichert. Es können gleichzeitig neue Zustände, mit geänderten Schildern erzeugt werden, ohne den gesamten Vektor zu kopieren. Für Anwendungsfälle wie diesen, bei denen recht einfache Objekte abgelegt werden sollen, auf die geteilt zugegriffen werden muss, lohnt sich eine persistente Umsetzung sicher.

Abschließend ist zu sagen, dass beim Einsatz von persistenten Datenstrukturen als Zustände von Anfang an darauf geachtet werden sollte, dass die einzelnen Komponenten als unveränderbare Objekte implementiert werden. Im Nachhinein die Implementierungen abzuändern wird bei zunehmender Projektgröße immer schwieriger. Dies liegt unter anderem daran, dass bei der Konvertierung von veränderbaren zu unveränderbaren Objekten nicht nur Interna der Objekte geändert werden, sondern auch die Verwendung von Nutzern angepasst werden muss. Je nach Anwendungsfall sind große Anpassungen in der Programmlogik nötig.

7.5.3 Wertung

Im momentanen Stand des A_2O Projekts kann es zu keinen Konflikten im WorldState kommen. Dies liegt an der Aufgabentrennung zwischen den einzelnen Threads. Auch wird pro Aufgabe nur ein Thread gestartet, welcher in einer Schleife neue Sensordaten verarbeitet. Allerdings würde durch die Implementierung des WorldStates als persistentes Objekt der Vorteil entstehen, dass unterschiedliche Threads an den gleichen Teilen des Zustandes gleichzeitig Modifikationen durchführen können. Anbieten würde dies sich beispielsweise für die Objekterkennung, welche über die Ausmaße und Position der Objekte auch Schilder erkennen könnte.

Da die einzelnen Arbeiterthreads im momentanen Stand auch keine Informationen aus dem bisherigen WorldState benutzten, um neue Informationen aus den Sensoren zu generieren, ist fraglich ob die Vorteile einer Implementierung mit persistenter Zustandsklasse überwiegen. Zwar muss in der Implementierung ohne persistenten Zustand manuell synchronisiert werden, da aber nur einige wenige Threads verwendet werden und die Übergänge leicht definierbar sind kommt es hier zu wenig Problemen bezüglich der Fehleranfälligkeit. Trotzdem war es sinnvoll eine Implementierung für ein bestehendes Problem in C++ umzusetzen, da vorher nicht abzusehende Probleme sowohl für die konkrete Implementierung als auch in der Verwendung von C++ erkannt und dokumentiert wurden.

8 Ausblick

Während bei der Bearbeitung des Themas persistente Datenstrukturen für asynchrone Umgebungen viele Erkenntnisse über die Praktikabilität der Verwendung und der Komplexität der Verwendung von persistenten Datenstrukturen gewonnen werden konnten, bleiben viele Fragestellungen offen.

Es wäre interessant das entwickelte Java Framework in einem realen Anwendungsfall zu benutzten. Hierbei wäre zu untersuchen, wie schwerwiegend die Einschränkungen des Frameworks sich auf die Benutzung auswirken. Vor allem müsste untersucht werden, ob die Probleme mit bedingten Modifikationen von Membern bei unveränderbaren Datenstrukturen weit verbreitet sind, und ob diese durch ein Verfahren automatisch erkannt werden können. Möglicherweise wäre eine Erkennung über eine automatisierte Analyse des Sourcecodes möglich.

Bei der Umsetzung des Mergeansatzes in C++, bei der zusätzlich zu den konzeptionellen Problemen noch Probleme mit der Implementierung hinzukommen, sollten außer dem Ansatz des manuellen Zusammenführens der Zustände, vor allem andere Herangehensweisen untersucht werden.

Ein andere Herangehensweise, die untersucht werden sollte, ist die Verwendung von confluent persistenten Datenstrukturen. Sie wären ausgezeichnet, um geteilte Zustände umzusetzen, die nach Bearbeitung wieder zusammengeführt werden sollen. Hier würde das Zusammenführen durch eine Meld-Operation der Datenstruktur selbst ermöglicht. Leider gibt es keine verfügbaren Implementierungen von Basisdatenstrukturen, mit denen diese Vorgehensweise untersucht werdenkann. Dies liegt unter anderem an der Komplexität der Implementierung. Zwar werden in verschiedenen Papers, wie für Sets und Maps in Liljenzin [2013] von Olle Liljenzin Verfahren dargestellt wie Datenstrukturen confluently persistent implementiert werden können, allerdings gibt es durch die erwarteten Performanzcharakteristiken wenige Programmierer die an einer praktischen Implementierung interessiert sind. Durch voll persistente Datenstrukturen konnten dagegen durch ihre gute Performanz auftretende Probleme in funktionalen Programmiersprachen gelöst werden. In diesem Bereich wäre es auch interessant die Ergebnisse die in Fiat und Kaplan [2001] vorgestellt werden, zu benutzen um einige confluently persistente Datenstrukturen umzusetzen. In ihrem Paper wird ein Verfahren vorgestellt, mit der beliebige Datenstrukturen mit beliebigen Operationen in confluent persistente Datenstrukturen transformiert

werden können. Hierbei werden große Einsparungen gegenüber dem naiven Node-Copying Verfahren erreicht.

Literaturverzeichnis

Adams 1993

Adams, Stephen: Functional Pearls Efficient sets—a balancing act. In: *Journal of functional programming* 3 (1993), Nr. 04, S. 553–561 3.1.2

Askitis und Sinha 2007

Askitis, Nikolas ; Sinha, Ranjan: HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In: Dobbie, Gillian (Hrsg.): *Thirtieth Australasian Computer Science Conference (ACSC2007)* Bd. 62. Ballarat Australia : ACS, 2007 (CRPIT), S. 97–105 3.1.1

Bagwell 2001

Bagwell, Phil: Ideal hash trees. 2001. – Forschungsbericht 2.2.6.2, 3.1.1

Crowley 1998

Crowley, Charles: Data structures for text sequences. In: *Computer Science Department, University of New Mexico, Date* (1998), S. 1–29 2.2.6.1

Driscoll u. a. 1986

Driscoll, James R. ; Sarnak, Neil ; Sleator, Daniel D. ; Tarjan, Robert E.: Making data structures persistent. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing* ACM, 1986, S. 109–121 2.2.5.1, 2.2.5.3, 2.2.5.4

Driscoll u. a. 1991

Driscoll, James R. ; Sleator, Daniel D. K. ; Tarjan, Robert E.: Fully Persistent Lists with Catenation. In: *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1991 (SODA '91). – ISBN 0–89791–376–0, 89–99 2.2.4.3

Farchi u. a. 2003

Farchi, E. ; Nir, Y. ; Ur, S.: Concurrent bug patterns and how to test them. In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003. – ISSN 1530–2075, S. 7 pp.– 5.1

Fiat und Kaplan 2001

Fiat, Amos ; Kaplan, Haim: Making data structures confluently persistent. In:

Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms Society for Industrial and Applied Mathematics, 2001, S. 537–546 8

Krukow

Krukow, Karl: *clj-ds GitHub*. https://github.com/krukow/clj-ds 3.1.1

Liljenzin 2013

Liljenzin, Olle: Confluently Persistent Sets and Maps. In: *CoRR* abs/1301.3388 (2013). http://arxiv.org/abs/1301.3388 8

Milewski

Milewski, Bartosz: *Okasaki GitHub*. https://github.com/BartoszMilewski/Okasaki 3.2.2

Okasaki 1999

Okasaki, Chris: *Purely functional data structures*. Cambridge University Press, 1999 3.1.1, 3.2.2

Oracle

Oracle: *The Java Tutorials*. http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html 2.2.1

PersistentVector

PersistentVector: *gorgone GitHub*. https://github.com/elnopintan/gorgone 3.2.3

Reps u. a. 1983

Reps, Thomas ; Teitelbaum, Tim ; Demers, Alan: Incremental Context-Dependent Analysis for Language-Based Editors. In: *ACM Trans. Program. Lang. Syst.* 5 (1983), Juli, Nr. 3, 449–477. http://dx.doi.org/10.1145/2166.357218. – DOI 10.1145/2166.357218. – ISSN 0164–0925 2.2.5.5

rotateright

rotateright: *Zoom Website*. http://www.rotateright.com/zoom 4.2

Russell und Norvig 2010

Russell, S.J. ; Norvig, P.: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010 (Prentice Hall series in artificial intelligence). https://books.google.de/books?id=8jZBksh-bUMC. – ISBN 9780136042594 4.3

Tanner

Tanner, Marc A.: *vis GitHub*. https://github.com/martanne/vis 2.2.6.1